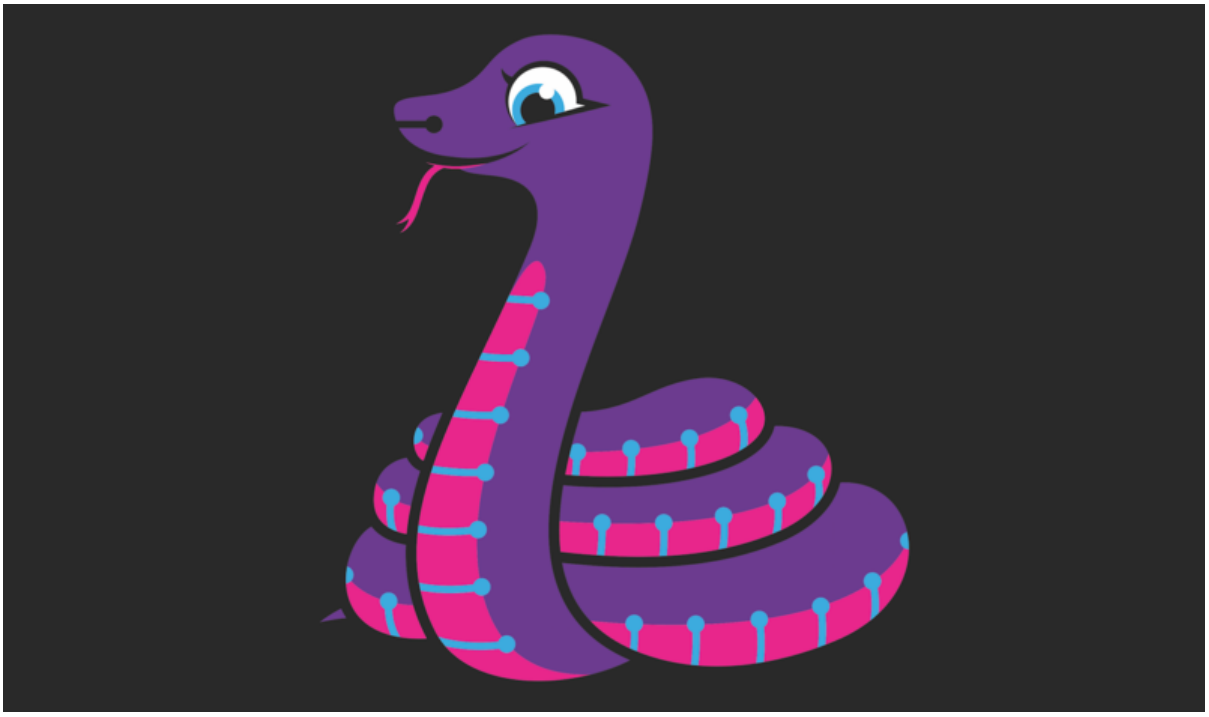
Welcome to CircuitPython!

Created by Kattni Rembor

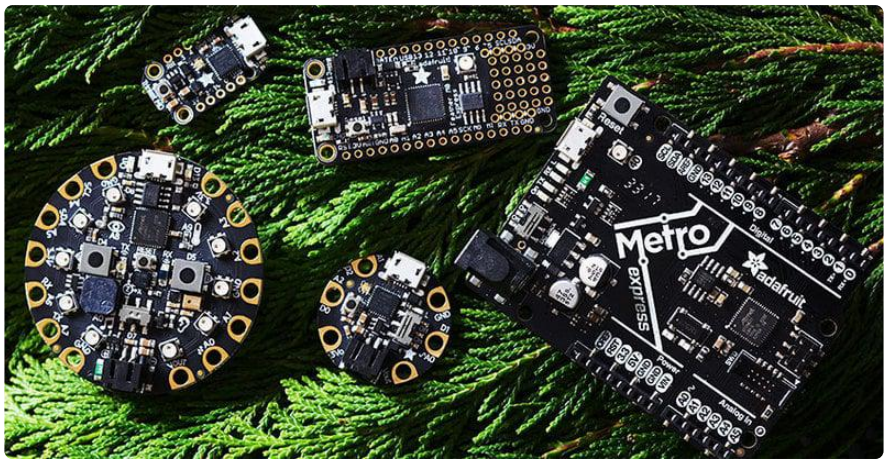


https://learn.adafruit.com/welcome-to-circuitpython

Last updated on 2023-10-13 10:34:03 AM EDT

Table of Contents

Welcome To CircuitPython



So, you've got a new CircuitPython compatible board. You plugged it in. Maybe it showed up as a disk drive called CIRCUITPY. Maybe it didn't! Either way, you need to know where to go from here. Well, this guide has you covered!

This guide will get you started with CircuitPython!

There are many amazing things about your new board. One of them is the ability to run CircuitPython. You may have seen that name on the Adafruit site () somewhere. Not sure what it is? This guide can help!

"But I've never coded in my life. There's no way I do it!" You absolutely can! CircuitPython is designed to help you learn from the ground up. If you're new to everything, this is the place to start!

This guide will walk you through how to get started with CircuitPython. You'll learn how to install CircuitPython, get updated to the newest version of CircuitPython, setup a serial connection, and edit your code. You'll learn some basics of how CircuitPython works, and about the CircuitPython libraries. You'll also find a list of frequently asked questions, and a series of troubleshooting steps if you run into any issues.

Welcome to CircuitPython!

What is CircuitPython?

CircuitPython is a programming language designed to simplify experimenting and learning to program on low-cost microcontroller boards. It makes getting started

easier than ever with no upfront desktop downloads needed. Once you get your board set up, open any text editor, and get started editing code. It's that simple.

CircuitPython is based on Python

Python is the fastest growing programming language. It's taught in schools and universities. It's a high-level programming language which means it's designed to be easier to read, write and maintain. It supports modules and packages which means it's easy to reuse your code for other projects. It has a built in interpreter which means there are no extra steps, like compiling, to get your code to work. And of course, Python is Open Source Software which means it's free for anyone to use, modify or improve upon.

CircuitPython adds hardware support to all of these amazing features. If you already have Python knowledge, you can easily apply that to using CircuitPython. If you have no previous experience, it's really simple to get started!

Why would I use CircuitPython?

CircuitPython is designed to run on microcontroller boards. A microcontroller board is a board with a microcontroller chip that's essentially an itty-bitty all-in-one computer. The board you're holding is a microcontroller board! CircuitPython is easy to use because all you need is that little board, a USB cable, and a computer with a USB connection. But that's only the beginning.

Other reasons to use CircuitPython include:

- You want to get up and running quickly. Create a file, edit your code, save the file, and it runs immediately. There is no compiling, no downloading and no uploading needed.
- You're new to programming. CircuitPython is designed with education in mind. It's easy to start learning how to program and you get immediate feedback from the board.
- Easily update your code. Since your code lives on the disk drive, you can edit it whenever you like, you can also keep multiple files around for easy experimentation.
- The serial console and REPL. These allow for live feedback from your code and interactive programming.
- File storage. The internal storage for CircuitPython makes it great for data-logging, playing audio clips, and otherwise interacting with files.
- Strong hardware support. CircuitPython has builtin support for microcontroller hardware features like digital I/O pins, hardware buses (UART, I2C, SPI), audio I/O, and other capabilities. There are also many libraries and drivers for sensors, breakout boards and other external components.
- It's Python! Python is the fastest-growing programming language. It's taught in schools and universities. CircuitPython is almost-completely compatible with Python. It simply adds hardware support.

This is just the beginning. CircuitPython continues to evolve, and is constantly being updated. Adafruit welcomes and encourages feedback from the community, and incorporate it into the development of CircuitPython. That's the core of the open source concept. This makes CircuitPython better for you and everyone who uses it!

Installing the Mu Editor

Mu is a simple code editor that works with the Adafruit CircuitPython boards. It's written in Python and works on Windows, MacOS, Linux and Raspberry Pi. The serial console is built right in so you get immediate feedback from your board's serial output!

Mu is our recommended editor - please use it (unless you are an experienced coder with a favorite editor already!).

Download and Install Mu

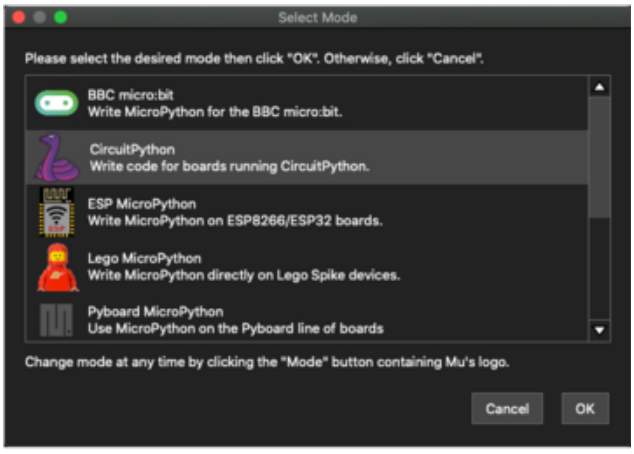
Download Mu from https://codewith.mu ().

Click the Download link for downloads and installation instructions.

Click Start Here to find a wealth of other information, including extensive tutorials and and how-to's.

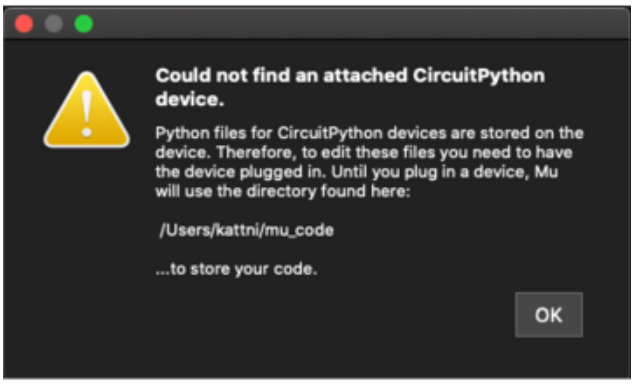
Windows users: due to the nature of MSI installers, please remove old versions of Mu before installing the latest version.

Starting Up Mu



The first time you start Mu, you will be prompted to select your 'mode' - you can always change your mind later. For now please select CircuitPython!

The current mode is displayed in the lower right corner of the window, next to the "gear" icon. If the mode says "Microbit" or something else, click the Mode button in the upper left, and then choose "CircuitPython" in the dialog box that appears.



Mu attempts to auto-detect your board on startup, so if you do not have a CircuitPython board plugged in with a CIRCUITPY drive available, Mu will inform you where it will store any code you save until you plug in a board.

To avoid this warning, plug in a board and ensure that the CIRCUITPY drive is mounted before starting Mu.

Using Mu

You can now explore Mu! The three main sections of the window are labeled below; the button bar, the text editor, and the serial console / REPL.

Now you're ready to code! Let's keep going...

Installing CircuitPython

As CircuitPython development continues and and there are new releases, Adafruit will stop supporting older releases. If you are running earlier versions of CircuitPython, you need to update to the latest. Generally Adafruit will support the last two major versions.

Some of the CircuitPython compatible boards come with CircuitPython installed. Others are CircuitPython-ready, but need to have it installed. As well, you may want to

update the version of CircuitPython already installed on your board. The steps are the same for installing and updating. This section will cover how to install or update CircuitPython on your board.

You only have to install CircuitPython ONCE. After that you are free to code all you like without going through this process again until it's time to upgrade!

Download the latest version!

The first thing you'll want to do is download the most recent version of CircuitPython.

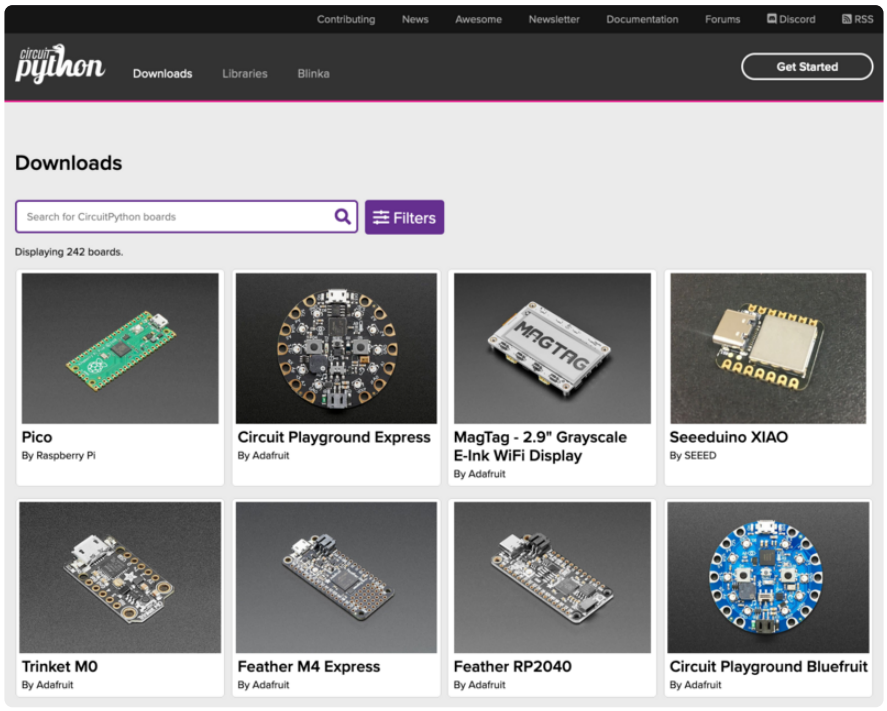
If you're already running CircuitPython, make sure you're running the latest version! If you're unsure, you can follow the steps below to ensure you have the latest version installed.

Always back up your code before installing or updating CircuitPython!

ALWAYS BACKUP YOUR CODE BEFORE INSTALLING OR UPDATING CIRCUITPYTHON. Most of the time, nothing will be removed from your board during the update, but it can happen. If you already have code on your board, be sure to back it up to your computer before following the steps below.

Download the latest software for your board by clicking the green button below to go to CircuitPython.org ().

Click here to download CircuitPython from CircuitPython.org



Next, you'll want to plug in your board using a known-good USB data cable. Make sure the USB cable is a data cable! There are some that work only for charging and can lead to a lot of frustration.

Windows 7 and 8.1 Drivers

If you're using Windows 7 or 8.1, you need to install a driver before plugging in your board.

If you're using Windows 7 or 8.1, check out the Windows 7 and 8.1 Drivers page () for details.

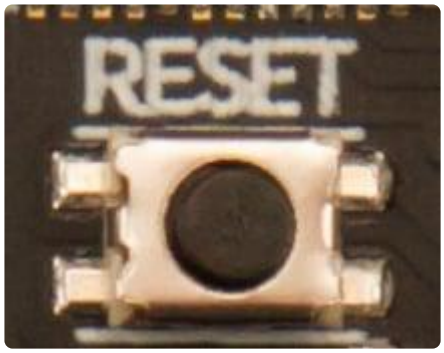
It is highly recommended that you upgrade to Windows 10.

Start the UF2 Bootloader

Nearly all CircuitPython boards ship with a bootloader called UF2 (USB Flashing Format) that makes installing and updating CircuitPython a quick and easy process. The bootloader is the mode your board needs to be in for the CircuitPython .uf2 file you downloaded to work. If the file you downloaded that matches the board name ends in uf2 then you want to continue with this section. However, if the file ends in .bin, you have to do a more complex installation - go to this page () for details.

For most boards:

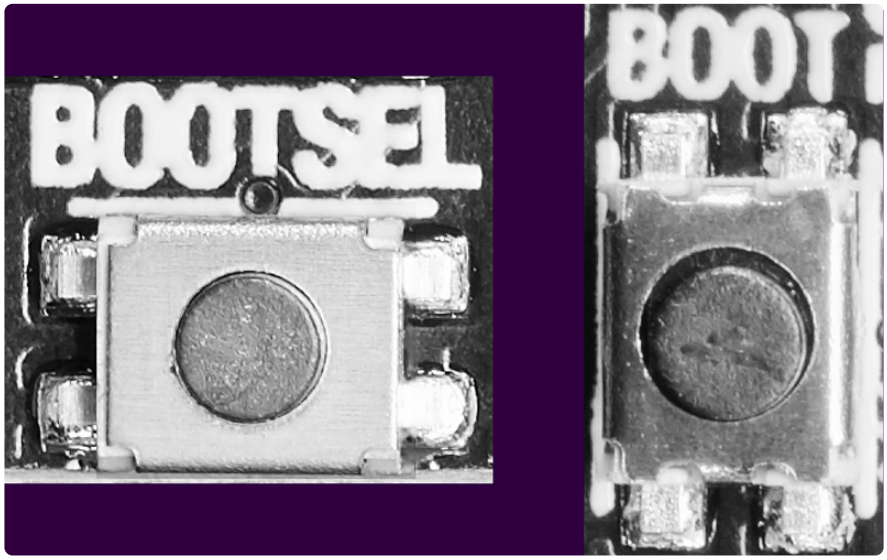
Find the reset button on your board. It's a small, black button, and on most of the boards, it will be the only button available. It is typically labeled RESET or RST on the board. (On Circuit Playground Express and Bluefruit, it's the smaller button located in the center of the board.)



Tap this button twice to enter the bootloader. If it doesn't work on the first try, don't be discouraged. The rhythm of the taps needs to be correct and sometimes it takes a few tries. If you have a Circuit Playground Express, and it's fresh-out-of-the-bag try pressing the button once.

For RP2040 boards:

You'll want to find two buttons on the RP2040 boards: reset and BOOTSEL/BOOT. The two buttons are the same size - small black buttons. Reset is typically labeled RESET or RST on the board. The boot button is labeled BOOTSEL or BOOT on the board.



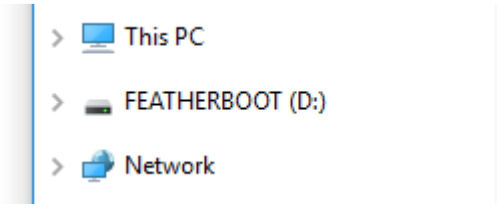
To enter the bootloader on an RP2040 board, you must hold down the boot select button, and while continuing to hold it, press and release the reset button. Continue to hold the boot select button until the bootloader drive appears.

Bootloader Mode

Once successful, the RGB status LED(s) on the board will flash red and then stay green. A new drive will show up on your computer.

For most boards:

The drive will be called boardnameBOOT where boardname is a reference to your specific board. For example, a basic Feather will have FEATHERBOOT and a Trinket will have TRINKETBOOT etc.



For RP2040 boards:

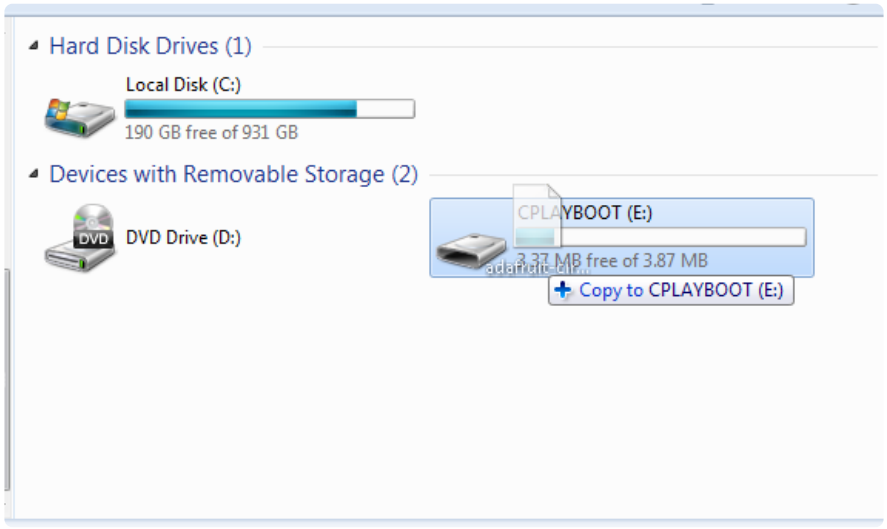
The drive will be called RPI-RP2 on all RP2040 boards.

Going forward, the bootloader drive will be referred to as the boot drive.

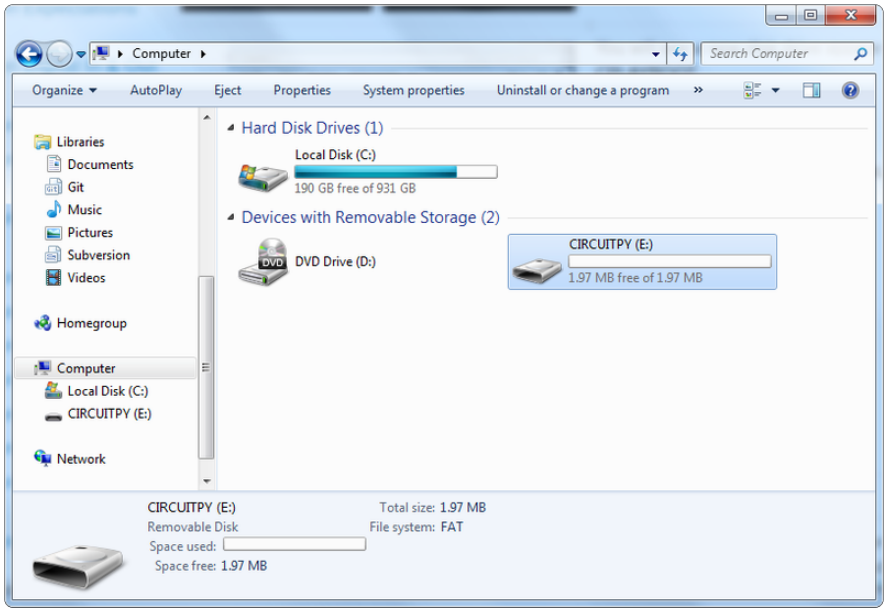
The board is now in bootloader mode! This is what you need to install or update CircuitPython.

Install CircuitPython

Now find the file you downloaded. Drag that file to the boot drive on your computer.



The lights should flash again, boot will disappear and a new drive will show up on your computer called CIRCUITPY.



Congratulations! You've successfully installed or updated CircuitPython!

What's the difference between CIRCUITPY and boardnameBOOT or RPI-RP2?

When you plug a CircuitPython board into your computer, your computer will see the board's flash memory as a USB flash drive where files can be stored. When you have

successfully installed CircuitPython, you'll see the CIRCUITPY drive. When you double-tap the reset button on most boards, you'll see the boardnameBOOT drive, or when hold boot select and tap reset for RP2040 boards, you'll see RPI-RP2. You can drag files to the boot drives and CIRCUITPY, but only CIRCUITPY will run your CircuitPython code.

Normally, when you drag a file to a mounted USB drive, the file copies to the drive and then is able to be seen in your file explorer. However, when you drag the CircuitPython UF2 file to the boot drive, it seems to disappear, and the drive disconnects. This is normal! The UF2 is essentially an installer file, and does not simply sit on the drive, but installs CircuitPython if the board is in bootloader mode (i.e. the boot drive).

You will be able to copy other files to the boot drive but they will not run or be accessible to CircuitPython. So make sure that once you're done installing CircuitPython, that you're dragging to and editing files on the CIRCUITPY drive!

Bootloader Drive Names

This list is not exhaustive, but should give you an idea what to look for in a bootloader drive name.

- Feather RP2040 = RPI-RP2
- QT Py RP2040 = RPI-RP2
- ItsyBitsy RP2040 = RPI-RP2
- Trinket M0 = TRINKETBOOT
- Gemma M0 = GEMMABOOT
- Circuit Playground Express = CPLAYBOOT
- ItsyBitsy M0 Express = ITSYBOOT
- ItsyBitsy M4 Express = ITSYM4BOOT
- Feather M0 Express = FEATHERBOOT
- Feather M4 Express = FEATHERBOOT
- Metro M0 Express = METROBOOT
- Metro M4 Express = METROM4BOOT
- Grand Central M4 Express = GCM4BOOT
- NeoTrelis M4 Express = TRELM4BOOT
- PyPortal, Pynt and Titano = PORTALBOOT

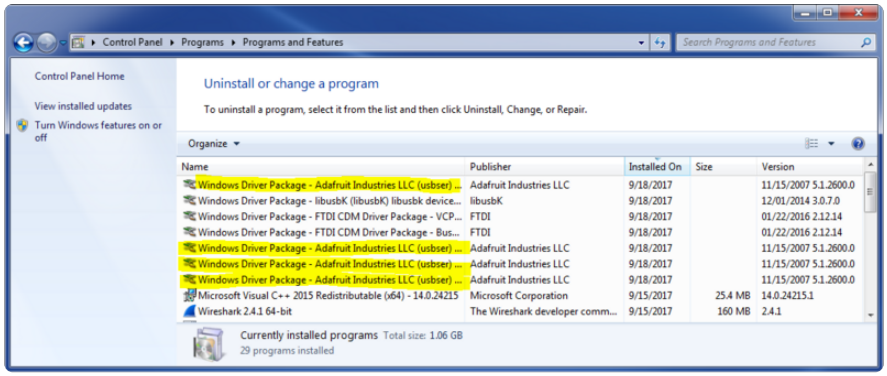
Windows 7 and 8.1 Drivers

If you're using Windows 7 or 8.1, you need to install a driver before plugging in your board.

If you're using Windows 7 or 8.1, you'll need to install a driver to use a CircuitPython-compatible board. You do not need to install drivers on Mac, Linux or Windows 10.

First, uninstall any older versions of the driver with the following steps:

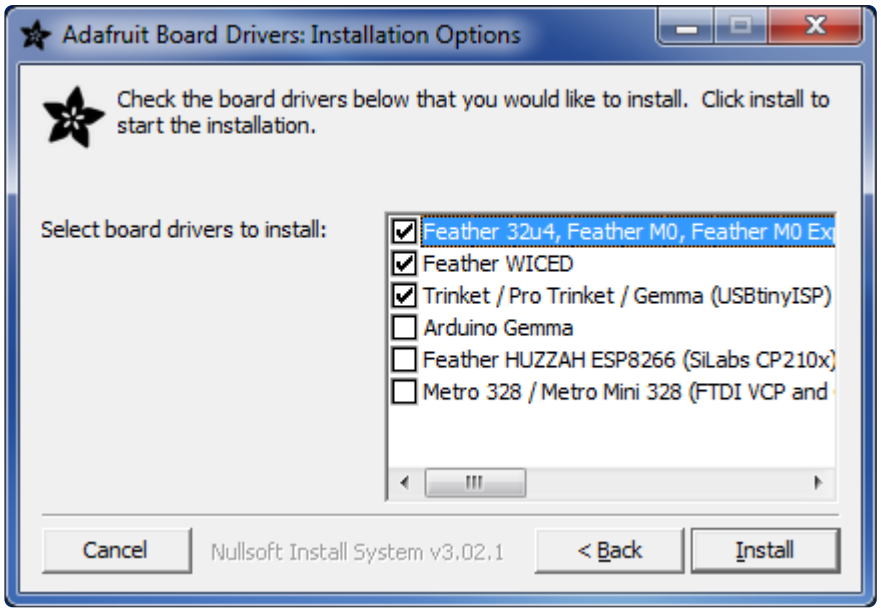
- Unplug any boards.
- Navigate to Uninstall or Change a Program by opening Control Panel->Programs->Uninstall a program.
- Uninstall everything named "Windows Driver Package - Adafruit Industries LLC ...".



Next, download the new 2.5.0.0 (or higher) Adafruit Windows Drivers Package using the following link:

Download Adafruit Windows 7/8.1 Driver Installer

When running the installer, you'll be shown a list of drivers to choose from. You can check and uncheck the boxes to choose which drivers to install.



The Windows Drivers installer was last updated in November 2020 (v2.5.0.0) . Windows 7/8.1 drivers for CircuitPython boards released since then, including RP2040 boards, are not available. The boards work fine on Windows 10 without extra drivers.

It is recommended that you upgrade to Windows 10. Windows 10 makes the drivers unnecessary and solves other problems.There are currently no plans to update the available drivers, given that Windows 7 is end-of-life. As of this writing, you can still upgrade for free from Windows 7 or 8.1 to Windows 10. See these () links ().

The CIRCUITPY Drive

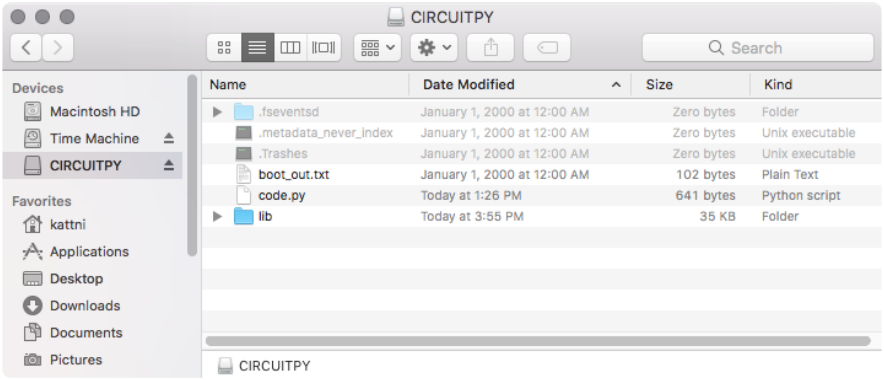
When CircuitPython finishes installing, or you plug a CircuitPython board into your computer with CircuitPython already installed, the board shows up on your computer as a USB drive called CIRCUITPY.

The CIRCUITPY drive is where your code and the necessary libraries and files will live. You can edit your code directly on this drive and when you save, it will run automatically. When you create and edit code, you'll save your code in a code.py file located on the CIRCUITPY drive. If you're following along with a Learn guide, you can paste the contents of the tutorial example into code.py on the CIRCUITPY drive and save it to run the example.

With a fresh CircuitPython install, on your CIRCUITPY drive, you'll find a code.py file containing `print("Hello World!")` and an empty lib folder. If your CIRCUITPY drive does not contain a code.py file, you can easily create one and save it to the drive. CircuitPython looks for code.py and executes the code within the file

automatically when the board starts up or resets. Following a change to the contents of CIRCUITPY, such as making a change to the code.py file, the board will reset, and the code will be run. You do not need to manually run the code. This is what makes it so easy to get started with your project and update your code!

Note that all changes to the contents of CIRCUITPY, such as saving a new file, renaming a current file, or deleting an existing file will trigger a reset of the board.



Boards Without CIRCUITPY

CircuitPython is available for some microcontrollers that do not support native USB. Those boards cannot present a CIRCUITPY drive. This includes boards using ESP32 or ESP32-C3 microcontrollers.

On these boards, there are alternative ways to transfer and edit files. You can use the Thonny editor (), which uses hidden commands sent to the REPL to read and write files. Or you can use the CircuitPython web workflow, introduced in Circuitpython 8. The web workflow provides browser-based WiFi access to the CircuitPython filesystem. These guides will help you with the web workflow:

- CircuitPython on ESP32 Quick Start ()
- CircuitPython Web Workflow Code Editor Quick Start ()

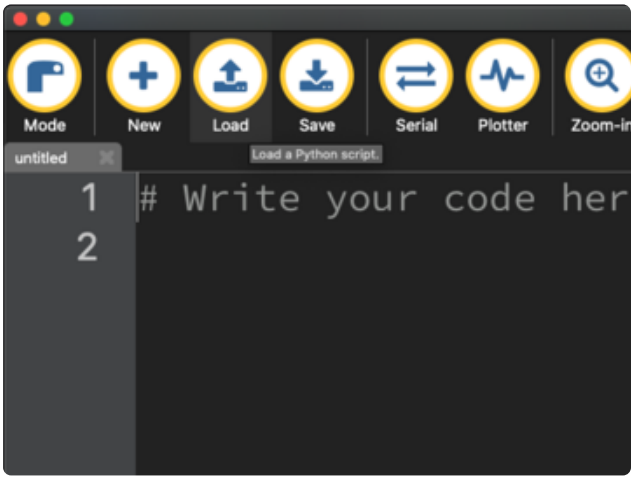
Creating and Editing Code

One of the best things about CircuitPython is how simple it is to get code up and running. This section covers how to create and edit your first CircuitPython program.

To create and edit code, all you'll need is an editor. There are many options. Adafruit strongly recommends using Mu! It's designed for CircuitPython, and it's really simple and easy to use, with a built in serial console!

If you don't or can't use Mu, there are a number of other editors that work quite well. The Recommended Editors page () has more details. Otherwise, make sure you do "Eject" or "Safe Remove" on Windows or "sync" on Linux after writing a file if you aren't using Mu. (This is not a problem on MacOS.)

Creating Code



Installing CircuitPython generates a code.py file on your CIRCUITPY drive. To begin your own program, open your editor, and load the code.py file from the CIRCUITPY drive.

If you are using Mu, click the Load button in the button bar, navigate to the CIRCUITPY drive, and choose code.py.

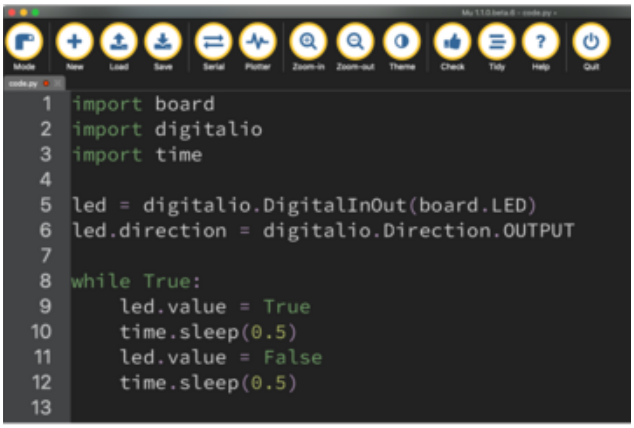
Copy and paste the following code into your editor:

```
import board
import digitalio
import time

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```

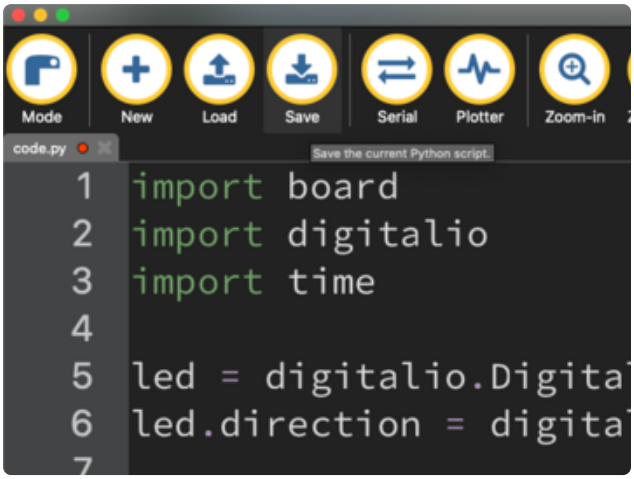
The KB2040, QT Py and the Trinkeys do not have a built-in little red LED! There is an addressable RGB NeoPixel LED. The above example will NOT work on the KB2040, QT Py or the Trinkeys!

If you're using a KB2040, QT Py or a Trinkey, please download the NeoPixel blink example ().

The NeoPixel blink example uses the onboard NeoPixel, but the time code is the same. You can use the linked NeoPixel Blink example to follow along with this guide page.



It will look like this. Note that under the `while True:` line, the next four lines begin with four spaces to indent them, and they're indented exactly the same amount. All the lines before that have no spaces before the text.



Save the code.py file on your CIRCUITPY drive.

The little LED should now be blinking. Once per half-second.

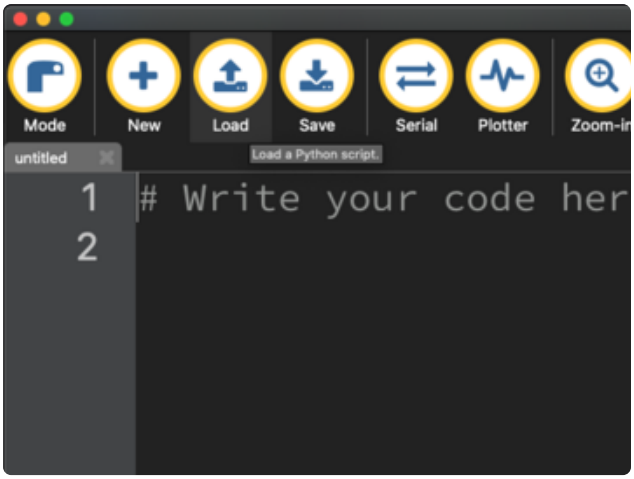
Congratulations, you've just run your first CircuitPython program!

On most boards you'll find a tiny red LED.

On the ItsyBitsy nRF52840, you'll find a tiny blue LED.

On QT Py M0, QT Py RP2040, and the Trinkey series, you will find only an RGB NeoPixel LED.

Editing Code



To edit code, open the code.py file on your CIRCUITPY drive into your editor.

Make the desired changes to your code. Save the file. That's it!

Your code changes are run as soon as the file is done saving.

There's one warning before you continue...

Don't click reset or unplug your board!

The CircuitPython code on your board detects when the files are changed or written and will automatically re-start your code. This makes coding very fast because you save, and it re-runs. If you unplug or reset the board before your computer finishes writing the file to your board, you can corrupt the drive. If this happens, you may lose the code you've written, so it's important to backup your code to your computer regularly.

There are a couple of ways to avoid filesystem corruption.

1. Use an editor that writes out the file completely when you save it.

Check out the Recommended Editors page () for details on different editing options.

If you are dragging a file from your host computer onto the CIRCUITPY drive, you still need to do step 2. Eject or Sync (below) to make sure the file is completely written.

2. Eject or Sync the Drive After Writing

If you are using one of our not-recommended-editors, not all is lost! You can still make it work.

On Windows, you can Eject or Safe Remove the CIRCUITPY drive. It won't actually eject, but it will force the operating system to save your file to disk. On Linux, use the sync command in a terminal to force the write to disk.

You also need to do this if you use Windows Explorer or a Linux graphical file manager to drag a file onto CIRCUITPY.

Oh No I Did Something Wrong and Now The CIRCUITPY Drive Doesn't Show Up!!!

> Don't worry! Corrupting the drive isn't the end of the world (or your board!). If this happens, follow the steps found on the Troubleshooting () page of every board guide to get your board up and running again.

Back to Editing Code...

Now! Let's try editing the program you added to your board. Open your code.py file into your editor. You'll make a simple change. Change the first `0.5` to `0.1`. The code should look like this:

```
import board
import digitalio
import time

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

while True:
    led.value = True
    time.sleep(0.1)
    led.value = False
    time.sleep(0.5)
```

Leave the rest of the code as-is. Save your file. See what happens to the LED on your board? Something changed! Do you know why?

You don't have to stop there! Let's keep going. Change the second `0.5` to `0.1` so it looks like this:

```
while True:
    led.value = True
```

```
    time.sleep(0.1)
    led.value = False
    time.sleep(0.1)
```

Now it blinks really fast! You decreased the both time that the code leaves the LED on and off!

Now try increasing both of the `0.1` to `1`. Your LED will blink much more slowly because you've increased the amount of time that the LED is turned on and off.

Well done! You're doing great! You're ready to start into new examples and edit them to see what happens! These were simple changes, but major changes are done using the same process. Make your desired change, save it, and get the results. That's really all there is to it!

Naming Your Program File

CircuitPython looks for a code file on the board to run. There are four options: code.txt, code.py, main.txt and main.py. CircuitPython looks for those files, in that order, and then runs the first one it finds. While code.py is the recommended name for your code file, it is important to know that the other options exist. If your program doesn't seem to be updating as you work, make sure you haven't created another code file that's being read instead of the one you're working on.

Exploring Your First CircuitPython Program

First, you'll take a look at the code you're editing.

Here is the original code again:

```
import board
import digitalio
import time

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```

Imports & Libraries

Each CircuitPython program you run needs to have a lot of information to work. The reason CircuitPython is so simple to use is that most of that information is stored in other files and works in the background. The files built into CircuitPython are called m odules, and the files you load separately are called libraries. Modules are built into CircuitPython. Libraries are stored on your CIRCUITPY drive in a folder called lib.

```
import board
import digitalio
import time
```

The `import` statements tells the board that you're going to use a particular library or module in your code. In this example, you imported three modules: `board`, `digitalio`, and `time`. All three of these modules are built into CircuitPython, so no separate library files are needed. That's one of the things that makes this an excellent first example. You don't need anything extra to make it work!

These three modules each have a purpose. The first one, `board`, gives you access to the hardware on your board. The second, `digitalio`, lets you access that hardware as inputs/outputs. The third, `time`, let's you control the flow of your code in multiple ways, including passing time by 'sleeping'.

Setting Up The LED

The next two lines setup the code to use the LED.

```
led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT
```

Your board knows the red LED as `LED`. So, you initialise that pin, and you set it to output. You set `led` to equal the rest of that information so you don't have to type it all out again later in our code.

Loop-de-loops

The third section starts with a `while` statement. `while True:` essentially means, "forever do the following:". `while True:` creates a loop. Code will loop "while" the condition is "true" (vs. false), and as `True` is never False, the code will loop forever. All code that is indented under `while True:` is "inside" the loop.

Inside our loop, you have four items:

```
while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```

First, you have `led.value = True`. This line tells the LED to turn on. On the next line, you have `time.sleep(0.5)`. This line is telling CircuitPython to pause running code for 0.5 seconds. Since this is between turning the led on and off, the led will be on for 0.5 seconds.

The next two lines are similar. `led.value = False` tells the LED to turn off, and `time.sleep(0.5)` tells CircuitPython to pause for another 0.5 seconds. This occurs between turning the led off and back on so the LED will be off for 0.5 seconds too.

Then the loop will begin again, and continue to do so as long as the code is running!

So, when you changed the first `0.5` to `0.1`, you decreased the amount of time that the code leaves the LED on. So it blinks on really quickly before turning off!

Great job! You've edited code in a CircuitPython program!

What Happens When My Code Finishes Running?

When your code finishes running, CircuitPython resets your microcontroller board to prepare it for the next run of code. That means any set up you did earlier no longer applies, and the pin states are reset.

For example, try reducing the code snippet above by eliminating the loop entirely, and replacing it with `led.value = True`. The LED will flash almost too quickly to see, and turn off. This is because the code finishes running and resets the pin state, and the LED is no longer receiving a signal.

To that end, most CircuitPython programs involve some kind of loop, infinite or otherwise.

What if I Don't Have the Loop?

If you don't have the loop, the code will run to the end and exit. This can lead to some unexpected behavior in simple programs like this since the "exit" also resets the state of the hardware. This is a different behavior than running commands via REPL. So if

you are writing a simple program that doesn't seem to work, you may need to add a loop to the end so the program doesn't exit.

The simplest loop would be:

```
while True:
    pass
```

And remember - you can press CTRL+C to exit the loop.

See also the Behavior section in the docs ().

Connecting to the Serial Console

One of the staples of CircuitPython (and programming in general!) is something called a "print statement". This is a line you include in your code that causes your code to output text. A print statement in CircuitPython (and Python) looks like this:

`print("Hello, world!")`

This line in your code.py would result in:

`Hello, world!`

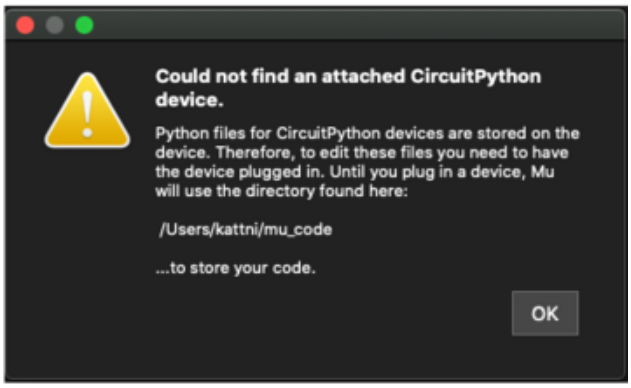
However, these print statements need somewhere to display. That's where the serial console comes in!

The serial console receives output from your CircuitPython board sent over USB and displays it so you can see it. This is necessary when you've included a print statement in your code and you'd like to see what you printed. It is also helpful for troubleshooting errors, because your board will send errors and the serial console will display those too.

The serial console requires an editor that has a built in terminal, or a separate terminal program. A terminal is a program that gives you a text-based interface to perform various tasks.

Are you using Mu?

If so, good news! The serial console is built into Mu and will autodetect your board making using the serial console really really easy.



First, make sure your CircuitPython board is plugged in.

If you open Mu without a board plugged in, you may encounter the error seen here, letting you know no CircuitPython board was found and indicating where your code will be stored until you plug in a board.

If you are using Windows 7, make sure you installed the drivers ().

Once you've opened Mu with your board plugged in, look for the Serial button in the button bar and click it.

The Mu window will split in two, horizontally, and display the serial console at the bottom.

If nothing appears in the serial console, it may mean your code is done running or has no print statements in it. Click into the serial console part of Mu, and press CTRL+D to reload.

Serial Console Issues or Delays on Linux

If you're on Linux, and are seeing multi-second delays connecting to the serial console, or are seeing "AT" and other gibberish when you connect, then the `modemmanager` service might be interfering. Just remove it; it doesn't have much use unless you're still using dial-up modems.

To remove `modemmanager`, type the following command at a shell:

```
sudo apt purge modemmanager
```

Setting Permissions on Linux

On Linux, if you see an error box something like the one below when you press the S erial button, you need to add yourself to a user group to have permission to connect to the serial console.

On Ubuntu and Debian, add yourself to the dialout group by doing:

```
sudo adduser $USER dialout
```

After running the command above, reboot your machine to gain access to the group. On other Linux distributions, the group you need may be different. See the Advanced Serial Console on Linux () for details on how to add yourself to the right group.

Using Something Else?

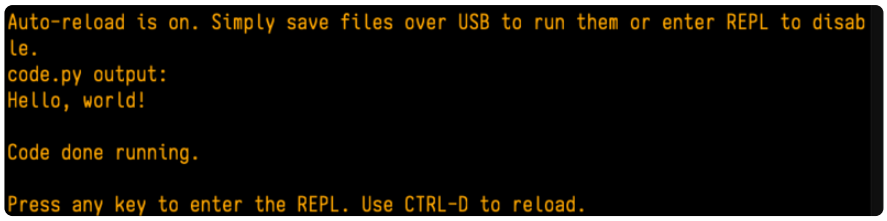
If you're not using Mu to edit, are using or if for some reason you are not a fan of its built in serial console, you can run the serial console from a separate program.

Windows requires you to download a terminal program. Check out the Advanced Serial Console on Windows page for more details. ()

MacOS has Terminal built in, though there are other options available for download. C heck the Advanced Serial Console on Mac page for more details. ()

Linux has a terminal program built in, though other options are available for download. Check the Advanced Serial Console on Linux page for more details. ()

Once connected, you'll see something like the following.



Interacting with the Serial Console

Once you've successfully connected to the serial console, it's time to start using it.

The code you wrote earlier has no output to the serial console. So, you're going to edit it to create some output.

Open your code.py file into your editor, and include a `print` statement. You can print anything you like! Just include your phrase between the quotation marks inside the parentheses. For example:

```
import board
import digitalio
import time

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

while True:
    print("Hello, CircuitPython!")
    led.value = True
    time.sleep(1)
    led.value = False
    time.sleep(1)
```

Save your file.

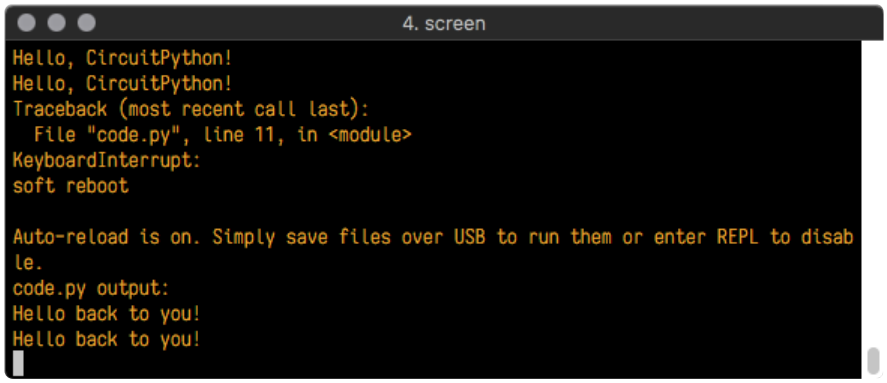
Now, let's go take a look at the window with our connection to the serial console.

Excellent! Our print statement is showing up in our console! Try changing the printed text to something else.

```
import board
import digitalio
import time
```

```
led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

while True:
    print("Hello back to you!")
    led.value = True
    time.sleep(1)
    led.value = False
    time.sleep(1)
```

Keep your serial console window where you can see it. Save your file. You'll see what the serial console displays when the board reboots. Then you'll see your new change!



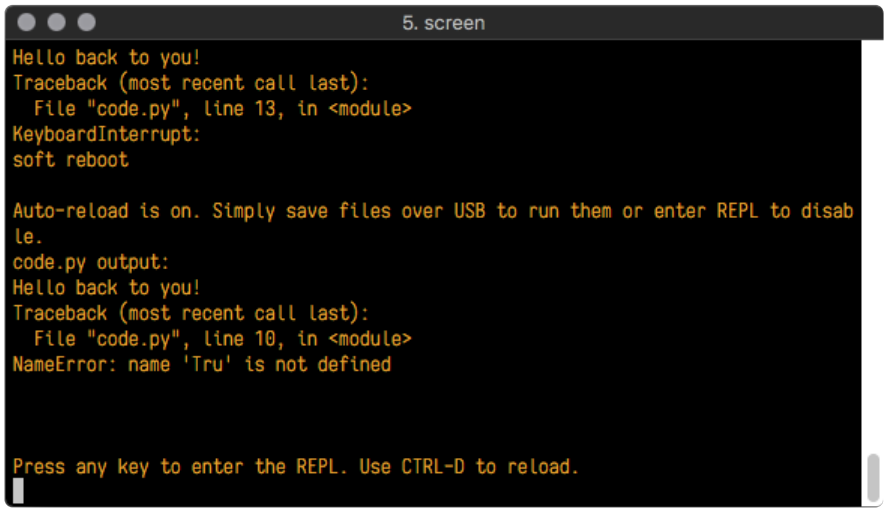
The `Traceback (most recent call last):` is telling you the last thing your board was doing before you saved your file. This is normal behavior and will happen every time the board resets. This is really handy for troubleshooting. Let's introduce an error so you can see how it is used.

Delete the `e` at the end of `True` from the line `led.value = True` so that it says `led.value = Tru`

```
import board
import digitalio
import time

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT

while True:
    print("Hello back to you!")
    led.value = Tru
    time.sleep(1)
    led.value = False
    time.sleep(1)
```

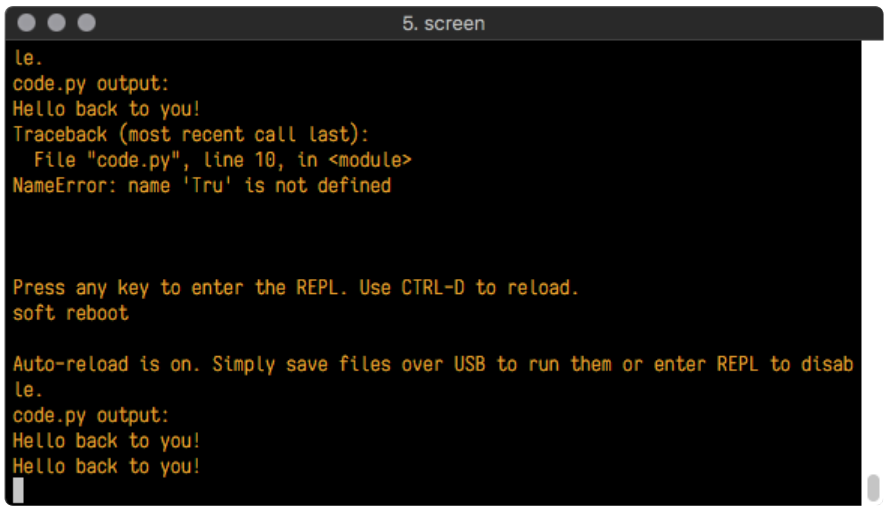
Save your file. You will notice that your red LED will stop blinking, and you may have a colored status LED blinking at you. This is because the code is no longer correct and can no longer run properly. You need to fix it!

Usually when you run into errors, it's not because you introduced them on purpose. You may have 200 lines of code, and have no idea where your error could be hiding. This is where the serial console can help. Let's take a look!



The `Traceback (most recent call last):` is telling you that the last thing it was able to run was `line 10` in your code. The next line is your error: `NameError: name 'Tru' is not defined`. This error might not mean a lot to you, but combined with knowing the issue is on line 10, it gives you a great place to start!

Go back to your code, and take a look at line 10. Obviously, you know what the problem is already. But if you didn't, you'd want to look at line 10 and see if you could figure it out. If you're still unsure, try googling the error to get some help. In this case, you know what to look for. You spelled True wrong. Fix the typo and save your file.



Nice job fixing the error! Your serial console is streaming and your red LED Is blinking again.

The serial console will display any output generated by your code. Some sensors, such as a humidity sensor or a thermistor, receive data and you can use print

statements to display that information. You can also use print statements for troubleshooting, which is called "print debugging". Essentially, if your code isn't working, and you want to know where it's failing, you can put print statements in various places to see where it stops printing.

The serial console has many uses, and is an amazing tool overall for learning and programming!

The REPL

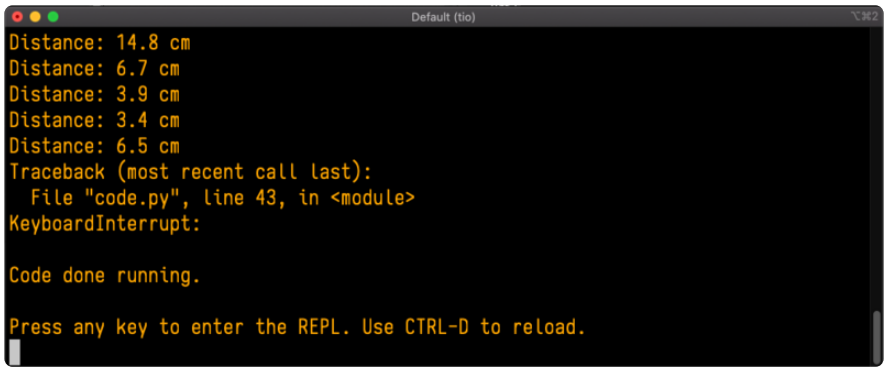
The other feature of the serial connection is the Read-Evaluate-Print-Loop, or REPL. The REPL allows you to enter individual lines of code and have them run immediately. It's really handy if you're running into trouble with a particular program and can't figure out why. It's interactive so it's great for testing new ideas.

Entering the REPL

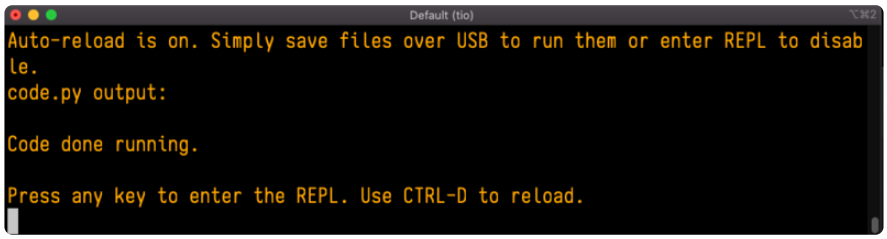
To use the REPL, you first need to be connected to the serial console. Once that connection has been established, you'll want to press CTRL+C.

If there is code running, in this case code measuring distance, it will stop and you'll see `Press any key to enter the REPL. Use CTRL-D to reload.` Follow those instructions, and press any key on your keyboard.

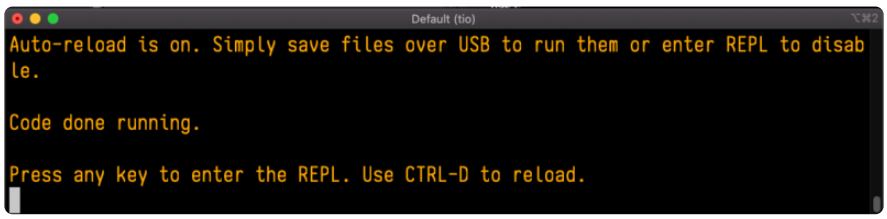
The `Traceback (most recent call last):` is telling you the last thing your board was doing before you pressed Ctrl + C and interrupted it. The `KeyboardInterrupt` is you pressing CTRL+C. This information can be handy when troubleshooting, but for now, don't worry about it. Just note that it is expected behavior.



If your code.py file is empty or does not contain a loop, it will show an empty output and `Code done running.`. There is no information about what your board was doing before you interrupted it because there is no code running.



If you have no code.py on your CIRCUITPY drive, you will enter the REPL immediately after pressing CTRL+C. Again, there is no information about what your board was doing before you interrupted it because there is no code running.



Regardless, once you press a key you'll see a `>>>` prompt welcoming you to the REPL!

If you have trouble getting to the `>>>` prompt, try pressing Ctrl + C a few more times.

The first thing you get from the REPL is information about your board.

```
Adafruit CircuitPython 7.0.0 on 2021-10-26; Adafruit Feather RP2040 with rp2040
```

This line tells you the version of CircuitPython you're using and when it was released. Next, it gives you the type of board you're using and the type of microcontroller the board uses. Each part of this may be different for your board depending on the versions you're working with.

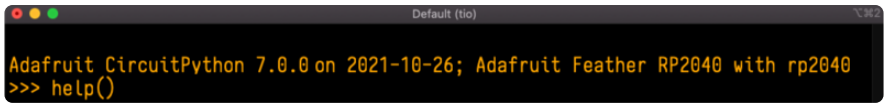
This is followed by the CircuitPython prompt.

```
>>>
```

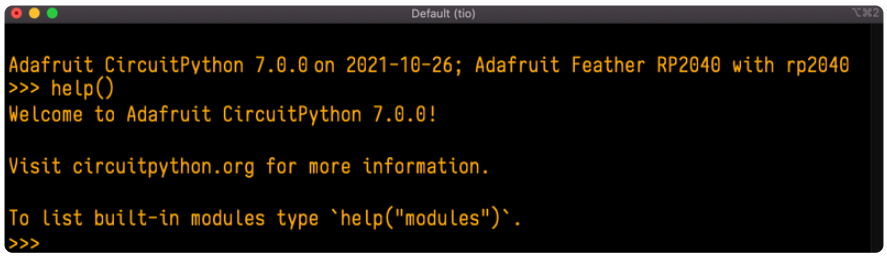
Interacting with the REPL

From this prompt you can run all sorts of commands and code. The first thing you'll do is run `help()`. This will tell you where to start exploring the REPL. To run code in the REPL, type it in next to the REPL prompt.

Type `help()` next to the prompt in the REPL.



Then press enter. You should then see a message.



First part of the message is another reference to the version of CircuitPython you're using. Second, a URL for the CircuitPython related project guides. Then... wait. What's this? `To list built-in modules type `help("modules")`.` Remember the modules you learned about while going through creating code? That's exactly what this is talking about! This is a perfect place to start. Let's take a look!

Type `help("modules")` into the REPL next to the prompt, and press enter.

This is a list of all the core modules built into CircuitPython, including `board`. Remember, `board` contains all of the pins on the board that you can use in your code. From the REPL, you are able to see that list!

Type `import board` into the REPL and press enter. It'll go to a new prompt. It might look like nothing happened, but that's not the case! If you recall, the `import` statement simply tells the code to expect to do something with that module. In this case, it's telling the REPL that you plan to do something with that module.

Next, type `dir(board)` into the REPL and press enter.

This is a list of all of the pins on your board that are available for you to use in your code. Each board's list will differ slightly depending on the number of pins available. Do you see `LED`? That's the pin you used to blink the red LED!

The REPL can also be used to run code. Be aware that any code you enter into the REPL isn't saved anywhere. If you're testing something new that you'd like to keep, make sure you have it saved somewhere on your computer as well!

Every programmer in every programming language starts with a piece of code that says, "Hello, World." You're going to say hello to something else. Type into the REPL:

```
print("Hello, CircuitPython!")
```

Then press enter.

That's all there is to running code in the REPL! Nice job!

You can write single lines of code that run stand-alone. You can also write entire programs into the REPL to test them. Remember that nothing typed into the REPL is saved.

There's a lot the REPL can do for you. It's great for testing new ideas if you want to see if a few new lines of code will work. It's fantastic for troubleshooting code by entering it one line at a time and finding out where it fails. It lets you see what modules are available and explore those modules.

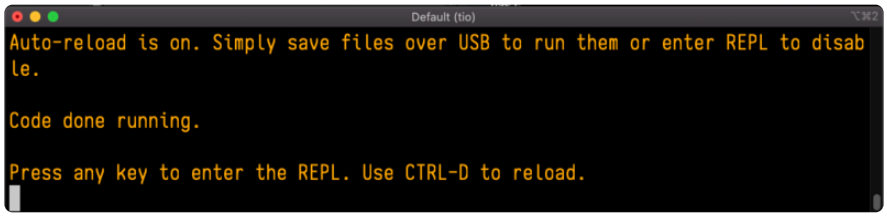
Try typing more into the REPL to see what happens!

Everything typed into the REPL is ephemeral. Once you reload the REPL or return to the serial console, nothing you typed will be retained in any memory space. So be sure to save any desired code you wrote somewhere else, or you'll lose it when you leave the current REPL instance!

Returning to the Serial Console

When you're ready to leave the REPL and return to the serial console, simply press CTRL+D. This will reload your board and reenter the serial console. You will restart the program you had running before entering the REPL. In the console window, you'll see any output from the program you had running. And if your program was affecting anything visual on the board, you'll see that start up again as well.

You can return to the REPL at any time!

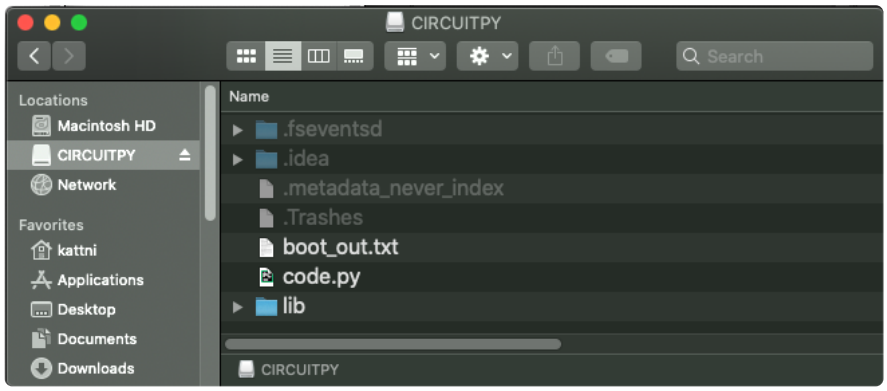


CircuitPython Libraries

As CircuitPython development continues and there are new releases, Adafruit will stop supporting older releases. Visit https://circuitpython.org/downloads to download the latest version of CircuitPython for your board. You must download the CircuitPython Library Bundle that matches your version of CircuitPython. Please update CircuitPython and then visit https://circuitpython.org/libraries to download the latest Library Bundle.

Each CircuitPython program you run needs to have a lot of information to work. The reason CircuitPython is so simple to use is that most of that information is stored in other files and works in the background. These files are called libraries. Some of them are built into CircuitPython. Others are stored on your CIRCUITPY drive in a folder called lib. Part of what makes CircuitPython so great is its ability to store code separately from the firmware itself. Storing code separately from the firmware makes it easier to update both the code you write and the libraries you depend.

Your board may ship with a lib folder already, it's in the base directory of the drive. If not, simply create the folder yourself. When you first install CircuitPython, an empty lib directory will be created for you.



CircuitPython libraries work in the same way as regular Python modules so the Python docs () are an excellent reference for how it all should work. In Python terms, you can place our library files in the lib directory because it's part of the Python path by default.

One downside of this approach of separate libraries is that they are not built in. To use them, one needs to copy them to the CIRCUITPY drive before they can be used. Fortunately, there is a library bundle.

The bundle and the library releases on GitHub also feature optimized versions of the libraries with the .mpy file extension. These files take less space on the drive and have a smaller memory footprint as they are loaded.

Due to the regular updates and space constraints, Adafruit does not ship boards with the entire bundle. Therefore, you will need to load the libraries you need when you begin working with your board. You can find example code in the guides for your board that depends on external libraries.

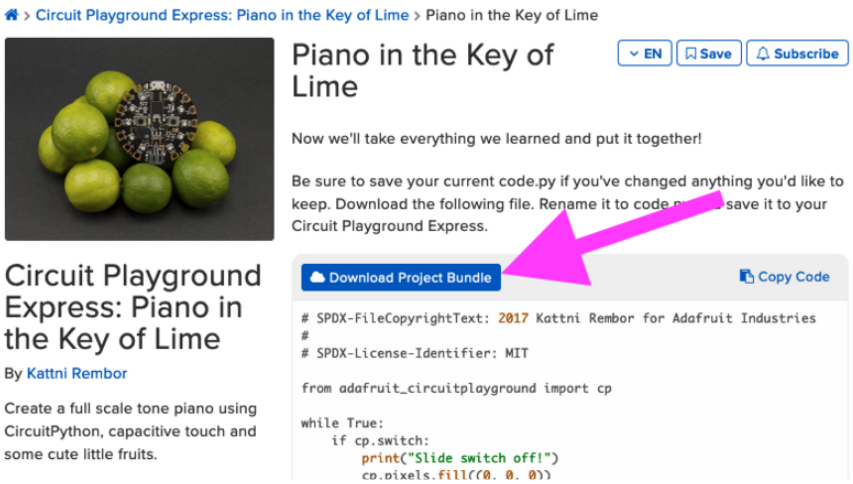
Either way, as you start to explore CircuitPython, you'll want to know how to get libraries on board.

The Adafruit Learn Guide Project Bundle

The quickest and easiest way to get going with a project from the Adafruit Learn System is by utilising the Project Bundle. Most guides now have a Download Project Bundle button available at the top of the full code example embed. This button downloads all the necessary files, including images, etc., to get the guide project up and running. Simply click, open the resulting zip, copy over the right files, and you're good to go!

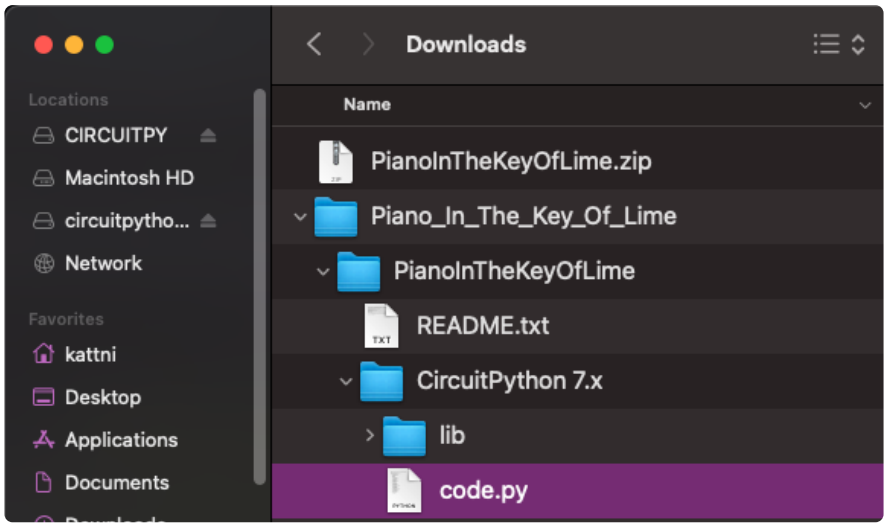
The first step is to find the Download Project Bundle button in the guide you're working on.

The Download Project Bundle button is only available on full demo code embedded from GitHub in a Learn guide. Code snippets will NOT have the button available.



When you copy the contents of the Project Bundle to your CIRCUITPY drive, it will replace all the existing content! If you don't want to lose anything, ensure you copy your current code to your computer before you copy over the new Project Bundle content!

The Download Project Bundle button downloads a zip file. This zip contains a series of directories, nested within which is the code.py, any applicable assets like images or audio, and the lib/ folder containing all the necessary libraries. The following zip was downloaded from the Piano in the Key of Lime guide.



The Piano in the Key of Lime guide was chosen as an example. That guide is specific to Circuit Playground Express, and cannot be used on all boards. Do not

expect to download that exact bundle and have it work on your non-CPX microcontroller.

When you open the zip, you'll find some nested directories. Navigate through them until you find what you need. You'll eventually find a directory for your CircuitPython version (in this case, 7.x). In the version directory, you'll find the file and directory you need: code.py and lib/. Once you find the content you need, you can copy it all over to your CIRCUITPY drive, replacing any files already on the drive with the files from the freshly downloaded zip.

In some cases, there will be other files such as audio or images in the same directory as code.py and lib/. Make sure you include all the files when you copy things over!

Once you copy over all the relevant files, the project should begin running! If you find that the project is not running as expected, make sure you've copied ALL of the project files onto your microcontroller board.

That's all there is to using the Project Bundle!

The Adafruit CircuitPython Library Bundle

Adafruit provides CircuitPython libraries for much of the hardware they provide, including sensors, breakouts and more. To eliminate the need for searching for each library individually, the libraries are available together in the Adafruit CircuitPython Library Bundle. The bundle contains all the files needed to use each library.

Downloading the Adafruit CircuitPython Library Bundle

You can download the latest Adafruit CircuitPython Library Bundle release by clicking the button below. The libraries are being constantly updated and improved, so you'll always want to download the latest bundle.

Match up the bundle version with the version of CircuitPython you are running. For example, you would download the 6.x library bundle if you're running any version of CircuitPython 6, or the 7.x library bundle if you're running any version of CircuitPython 7, etc. If you mix libraries with major CircuitPython versions, you will get incompatible mpy errors due to changes in library interfaces possible during major version changes.

Click to visit circuitpython.org for the latest Adafruit CircuitPython Library Bundle

Download the bundle version that matches your CircuitPython firmware version. If you don't know the version, check the version info in boot_out.txt file on the CIRCUITPY drive, or the initial prompt in the CircuitPython REPL. For example, if you're running v7.0.0, download the 7.x library bundle.

There's also a py bundle which contains the uncompressed python files, you probably don't want that unless you are doing advanced work on libraries.

The CircuitPython Community Library Bundle

The CircuitPython Community Library Bundle is made up of libraries written and provided by members of the CircuitPython community. These libraries are often written when community members encountered hardware not supported in the Adafruit Bundle, or to support a personal project. The authors all chose to submit these libraries to the Community Bundle make them available to the community.

These libraries are maintained by their authors and are not supported by Adafruit. As you would with any library, if you run into problems, feel free to file an issue on the GitHub repo for the library. Bear in mind, though, that most of these libraries are supported by a single person and you should be patient about receiving a response. Remember, these folks are not paid by Adafruit, and are volunteering their personal time when possible to provide support.

Downloading the CircuitPython Community Library Bundle

You can download the latest CircuitPython Community Library Bundle release by clicking the button below. The libraries are being constantly updated and improved, so you'll always want to download the latest bundle.

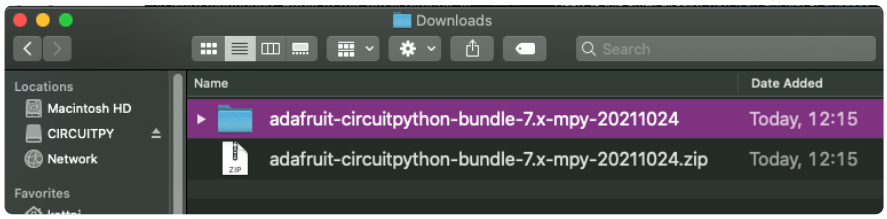
Click for the latest CircuitPython Community Library Bundle release

The link takes you to the latest release of the CircuitPython Community Library Bundle on GitHub. There are multiple versions of the bundle available. Download the bundle version that matches your CircuitPython firmware version. If you don't know

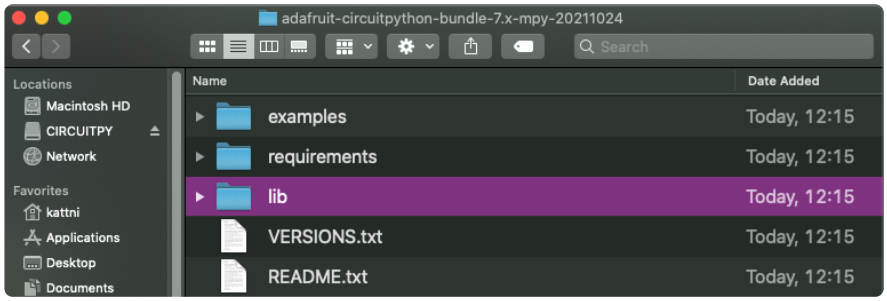
the version, check the version info in boot_out.txt file on the CIRCUITPY drive, or the initial prompt in the CircuitPython REPL. For example, if you're running v7.0.0, download the 7.x library bundle.

Understanding the Bundle

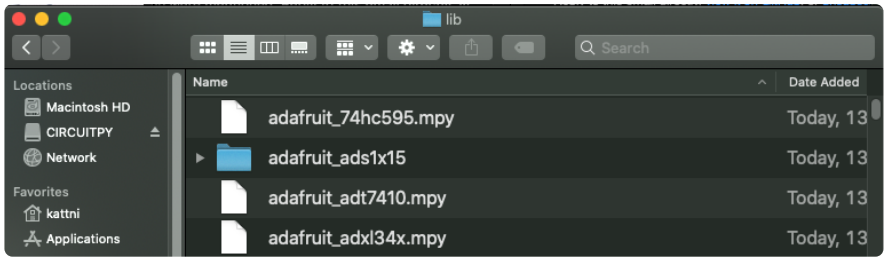
After downloading the zip, extract its contents. This is usually done by double clicking on the zip. On Mac OSX, it places the file in the same directory as the zip.



Open the bundle folder. Inside you'll find two information files, and two folders. One folder is the lib bundle, and the other folder is the examples bundle.



Now open the lib folder. When you open the folder, you'll see a large number of .mpy files, and folders.

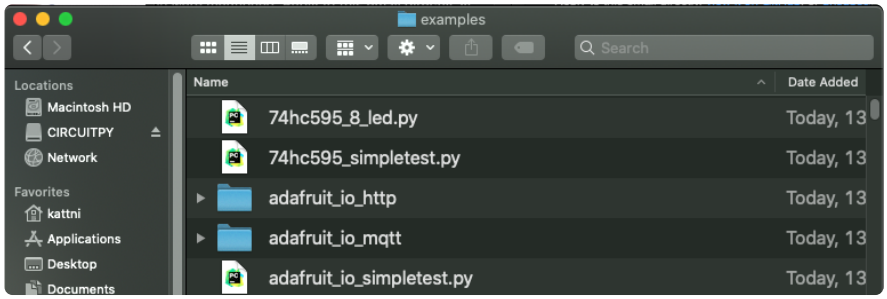


Example Files

All example files from each library are now included in the bundles in an examples directory (as seen above), as well as an examples-only bundle. These are included for two main reasons:

- Allow for quick testing of devices.

- Provide an example base of code, that is easily built upon for individualized purposes.



Copying Libraries to Your Board

First open the lib folder on your CIRCUITPY drive. Then, open the lib folder you extracted from the downloaded zip. Inside you'll find a number of folders and .mpy files. Find the library you'd like to use, and copy it to the lib folder on CIRCUITPY.

If the library is a directory with multiple .mpy files in it, be sure to copy the entire folder to CIRCUITPY/lib.

This also applies to example files. Open the examples folder you extracted from the downloaded zip, and copy the applicable file to your CIRCUITPY drive. Then, rename it to code.py to run it.

If a library has multiple .mpy files contained in a folder, be sure to copy the entire folder to CIRCUITPY/lib.

Understanding Which Libraries to Install

You now know how to load libraries on to your CircuitPython-compatible microcontroller board. You may now be wondering, how do you know which libraries you need to install? Unfortunately, it's not always straightforward. Fortunately, there is an obvious place to start, and a relatively simple way to figure out the rest. First up: the best place to start.

When you look at most CircuitPython examples, you'll see they begin with one or more `import` statements. These typically look like the following:

- `import library_or_module`

However, `import` statements can also sometimes look like the following:

- `from library_or_module import name`
- `from library_or_module.subpackage import name`
- `from library_or_module import name as local_name`

They can also have more complicated formats, such as including a `try` / `except` block, etc.

The important thing to know is that an `import` statement will always include the name of the module or library that you're importing.

Therefore, the best place to start is by reading through the `import` statements.

Here is an example import list for you to work with in this section. There is no setup or other code shown here, as the purpose of this section involves only the import list.

```
import time
import board
import neopixel
import adafruit_lis3dh
import usb_hid
from adafruit_hid.consumer_control import ConsumerControl
from adafruit_hid.consumer_control_code import ConsumerControlCode
```

Keep in mind, not all imported items are libraries. Some of them are almost always built-in CircuitPython modules. How do you know the difference? Time to visit the REPL.

In the Interacting with the REPL section () on The REPL page () in this guide, the `help("modules")` command is discussed. This command provides a list of all of the built-in modules available in CircuitPython for your board. So, if you connect to the serial console on your board, and enter the REPL, you can run `help("modules")` to see what modules are available for your board. Then, as you read through the `import` statements, you can, for the purposes of figuring out which libraries to load, ignore the statement that import modules.

The following is the list of modules built into CircuitPython for the Feather RP2040. Your list may look similar or be anything down to a significant subset of this list for smaller boards.

```
>>> help("modules")
__main__          board             micropython       storage
_bleio            builtins          msgpack           struct
adafruit_bus_device                 busio             neopixel_write    supervisor
adafruit_pixelbuf collections       onewireio         synthio
aesio             countio           os                sys
alarm             digitalio         paralleldisplay   terminalio
analogio          displayio         pulseio           time
array             errno             pwmio             touchio
atexit            fontio            qrio              traceback
audiobusio        framebufferio     rainbowio         ulab
audiocore         gc                random            usb_cdc
audiomixer        getpass           re                usb_hid
audiomp3          imagecapture      rgbmatrix         usb_midi
audiopwmio        io                rotaryio          vectorio
binascii          json              rp2pio            watchdog
bitbangio         keypad            rtc
bitmaptools       math              sdcardio
bitops            microcontroller   sharpdisplay
```

Now that you know what you're looking for, it's time to read through the import statements. The first two, `time` and `board`, are on the modules list above, so they're built-in.

The next one, `neopixel`, is not on the module list. That means it's your first library! So, you would head over to the bundle zip you downloaded, and search for neopixel. There is a neopixel.mpy file in the bundle zip. Copy it over to the lib folder on your CI RCUITPY drive. The following one, `adafruit_lis3dh`, is also not on the module list. Follow the same process for adafruit_lis3dh, where you'll find adafruit_lis3dh.mpy, and copy that over.

The fifth one is `usb_hid`, and it is in the modules list, so it is built in. Often all of the built-in modules come first in the import list, but sometimes they don't! Don't assume that everything after the first library is also a library, and verify each import with the modules list to be sure. Otherwise, you'll search the bundle and come up empty!

The final two imports are not as clear. Remember, when `import` statements are formatted like this, the first thing after the `from` is the library name. In this case, the library name is `adafruit_hid`. A search of the bundle will find an adafruit_hid folder. When a library is a folder, you must copy the entire folder and its contents as it is in the bundle to the lib folder on your CIRCUITPY drive. In this case, you would copy the entire adafruit_hid folder to your CIRCUITPY/lib folder.

Notice that there are two imports that begin with `adafruit_hid`. Sometimes you will need to import more than one thing from the same library. Regardless of how many times you import the same library, you only need to load the library by copying over the adafruit_hid folder once.

That is how you can use your example code to figure out what libraries to load on your CircuitPython-compatible board!

There are cases, however, where libraries require other libraries internally. The internally required library is called a dependency. In the event of library dependencies, the easiest way to figure out what other libraries are required is to connect to the serial console and follow along with the `ImportError` printed there. The following is a very simple example of an `ImportError`, but the concept is the same for any missing library.

Example: `ImportError` Due to Missing Library

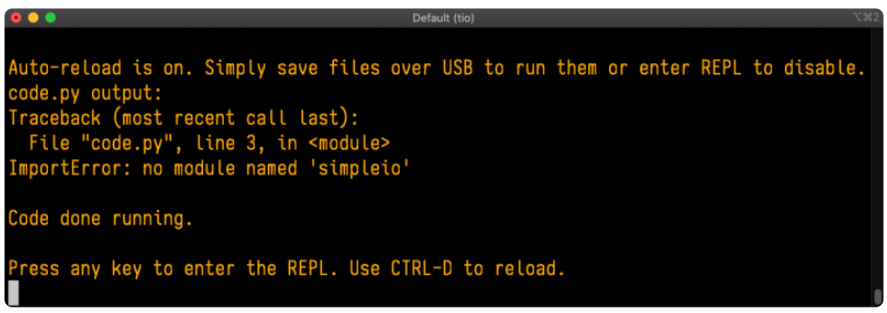
If you choose to load libraries as you need them, or you're starting fresh with an existing example, you may end up with code that tries to use a library you haven't yet loaded. This section will demonstrate what happens when you try to utilise a library that you don't have loaded on your board, and cover the steps required to resolve the issue.

This demonstration will only return an error if you do not have the required library loaded into the lib folder on your CIRCUITPY drive.

Let's use a modified version of the Blink example.

```
import board
import time
import simpleio

led = simpleio.DigitalOut(board.LED)

while True:
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
```

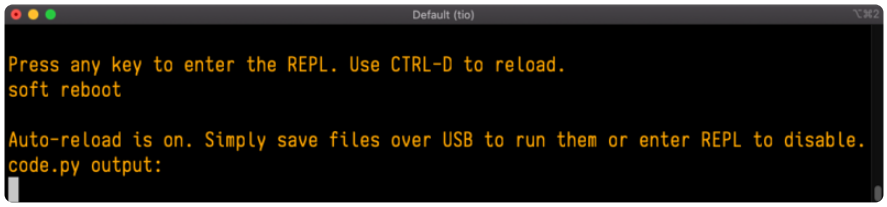
Save this file. Nothing happens to your board. Let's check the serial console to see what's going on.



You have an `ImportError`. It says there is `no module named 'simpleio'`. That's the one you just included in your code!

Click the link above to download the correct bundle. Extract the lib folder from the downloaded bundle file. Scroll down to find simpleio.mpy. This is the library file you're looking for! Follow the steps above to load an individual library file.

The LED starts blinking again! Let's check the serial console.



No errors! Excellent. You've successfully resolved an `ImportError`!

If you run into this error in the future, follow along with the steps above and choose the library that matches the one you're missing.

Library Install on Non-Express Boards

If you have an M0 non-Express board such as Trinket M0, Gemma M0, QT Py M0, or one of the M0 Trinkeys, you'll want to follow the same steps in the example above to install libraries as you need them. Remember, you don't need to wait for an `ImportError` if you know what library you added to your code. Open the library bundle you downloaded, find the library you need, and drag it to the lib folder on your CIRCUITPY drive.

You can still end up running out of space on your M0 non-Express board even if you only load libraries as you need them. There are a number of steps you can use to try to resolve this issue. You'll find suggestions on the Troubleshooting page ().

Updating CircuitPython Libraries and Examples

Libraries and examples are updated from time to time, and it's important to update the files you have on your CIRCUITPY drive.

To update a single library or example, follow the same steps above. When you drag the library file to your lib folder, it will ask if you want to replace it. Say yes. That's it!

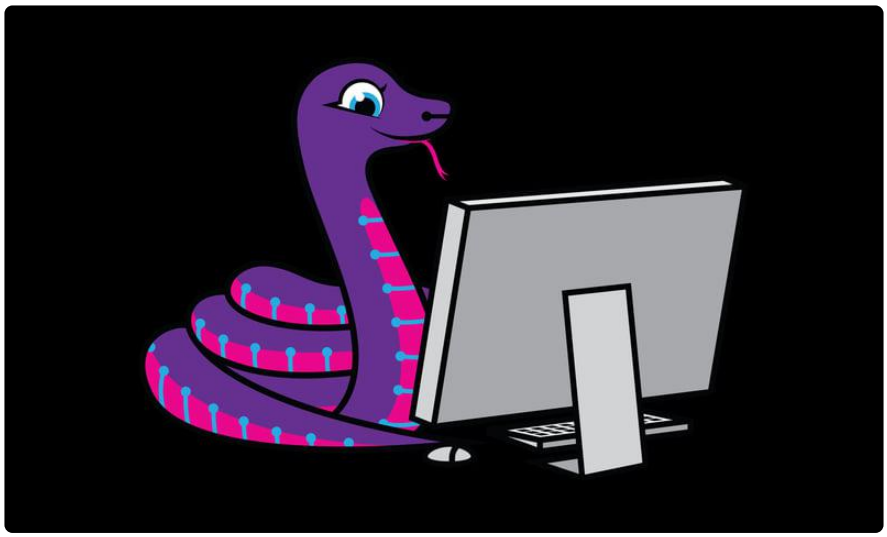
A new library bundle is released every time there's an update to a library. Updates include things like bug fixes and new features. It's important to check in every so often to see if the libraries you're using have been updated.

CircUp CLI Tool

There is a command line interface (CLI) utility called CircUp () that can be used to easily install and update libraries on your device. Follow the directions on the install page within the CircUp learn guide (). Once you've got it installed you run the command `circup update` in a terminal to interactively update all libraries on the connected CircuitPython device. See the usage page in the CircUp guide () for a full list of functionality

---

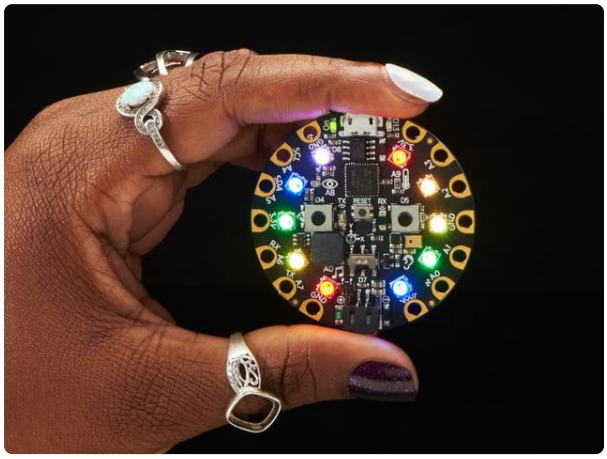
CircuitPython Hardware



Perhaps you're new to electronics and programming. Maybe you're a pro, but you've never worked with CircuitPython. Working with CircuitPython is super simple, but requires that you have a CircuitPython compatible microcontroller. Some boards are better than others for getting started than others. So which one do you choose?

This page contains all of the beginner CircuitPython compatible boards and a bit about the features of each one. You're ready to get started, now to figure out what features will work best for you!

Circuit Playground Express

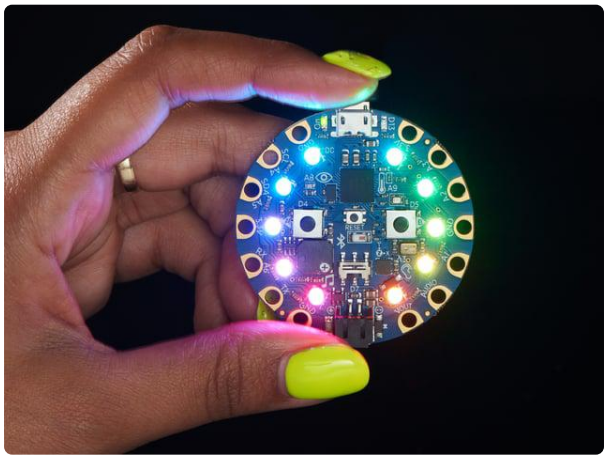


Circuit Playground Express
Circuit Playground Express is the next step towards a perfect introduction to electronics and programming. We've taken the original Circuit Playground Classic and...
https://www.adafruit.com/product/3333

The Adafruit Circuit Playground Express, running the ATSAMD21 microcontroller, is the perfect board for beginners to electronics and programming. It has tons built in: ten RGB NeoPixel LEDs, an accelerometer, temperature, light and sound sensors, mini speaker, two buttons, a slide switch, IR transmitter and receiver, little red LED, and eight alligator-clip friendly GPIO pads, seven of which can act as touch pads.

This is an excellent beginner choice that does not require any soldering.

Circuit Playground Bluefruit



Circuit Playground Bluefruit - Bluetooth Low Energy
Circuit Playground Bluefruit is our third board in the Circuit Playground series, another step towards a perfect introduction to electronics and programming. We've...
https://www.adafruit.com/product/4333

The Adafruit Circuit Playground Bluefruit, running the Bluetooth LE capabile nRF52840, is an all-in-one board designed to get you started with programming an electronics. It's loaded with all kinds of LEDs, sensors and inputs, including an accelerometer, light, temperature, and sound sensors, touch pads, buttons, switch, NeoPixel LEDs, speaker, and more. Additional capabilities can be added via the alligator clip friendly pads.

This is an excellent beginner choice that does not require any soldering.

Circuit Playground Express or Bluefruit?

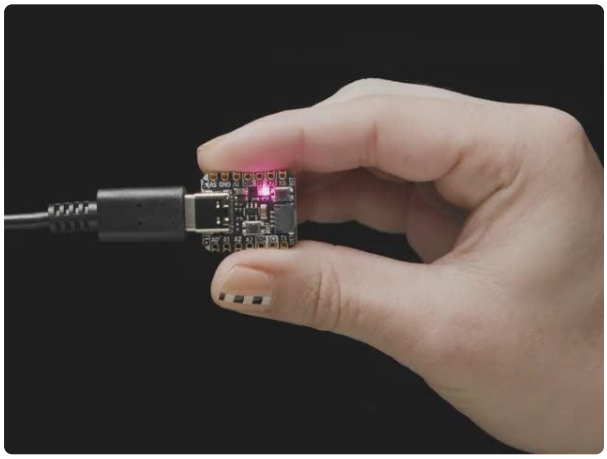
There are two great Circuit Playground options for using CircuitPython. So which one is better for you? You'll be fine with either, but here a few key differences that may drive you one way or the other.

- The Circuit Playground Express has more support in MakeCode () / Code.org and the SAMD21 is a more universal processor. Theres a lot more example codes for it. However, it's an older chip so not as powerful and of course, does not have Bluetooth LE wireless
- The Circuit Playground Bluefruit does not have an IR transceiver (for remote control projects) because it has BLE instead. The processor is more powerful, has tons more memory which comes in handy with CircuitPython projects as they grow. However, its a newer chip, and does not have full MakeCode support, Code.org CS Discoveries support, or as many projects.
  It also has Bluetooth LE connectivity so it can wirelessly connect to computers, phones, tablets and other devices.

The original ATmega 32u4 based Circuit Playground (Renamed the Circuit Playground Classic) is NOT CircuitPython compatible. Use the Express or Bluefruit versions.

QT Py RP2040

Easily interface to STEMMA QT / Qwiic sensors and breakouts.

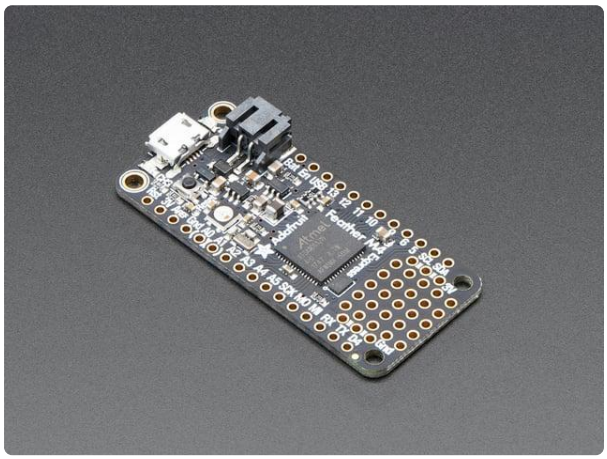


Adafruit QT Py RP2040
What a cutie pie! Or is it... a QT Py? This diminutive dev board comes with one of our new favorite chip, the RP2040. It's been made famous in the new
https://www.adafruit.com/product/4900

The Adafruit QT Py uses an RP2040 microcontroller, has 11 GPIO pins, a built in RGB NeoPixel LED, and a USB Type C connector, in the same size, form factor and pinout as the Seeed Xiao. There is 8MB of flash for CircuitPython and file storage. The built in STEMMA QT connector makes it super simple to use any of the available STEMMA QT sensors and breakouts () with no soldering required.

This is a good choice if you want an easy way to interface to STEMMA QT / Qwiic connected sensors without a lot of expense

Feather M4 Express



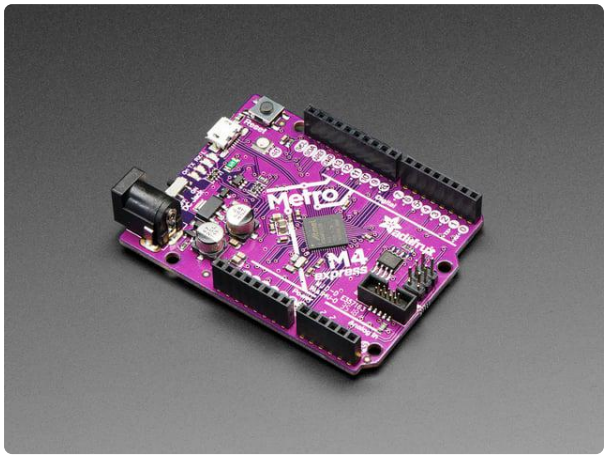
Adafruit Feather M4 Express - Featuring ATSAMD51

It's what you've been waiting for, the Feather M4 Express featuring ATSAMD51. This Feather is fast like a swift, smart like an owl, strong like a ox-bird (it's half ox,... https://www.adafruit.com/product/3857

The Adafruit Feather M4 Express is running the ATSAMD51 microcontroller. With some light soldering, this board enables you to use the many FeatherWings available (), which provides a ton of possibilities without a lot of wiring necessary. It has 21 GPIO pins, a little red LED and an RGB NeoPixel LED. You can power it with and charge a lipoly battery.

A good choice if you want access to Feather ecosystem and its many FeatherWing add on boards. The board is very powerful and runs CircuitPython very nicely

Metro M4



Adafruit Metro M4 feat. Microchip ATSAMD51

Are you ready? Really ready? Cause here comes the fastest, most powerful Metro ever. The Adafruit Metro M4 featuring the Microchip ATSAMD51. This...

https://www.adafruit.com/product/3382

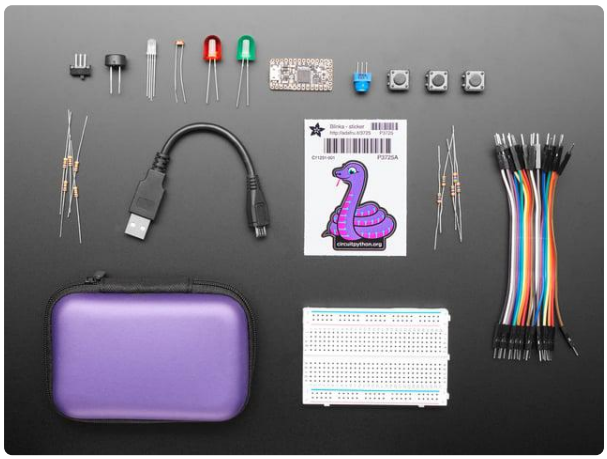
The Adafruit Metro M4, featuring the ATSAMD51 microcontroller, is a development board in the Metro form factor. It boasts 25 GPIO pins, along with four indicator LEDs and an RGB NeoPixel LED. It can be powered via USB using the micro USB connector, or through the DC jack. The great thing about this board is that it's compatible with Arduino Shields (), meaning there are a ton of possibilities available without a lot of wiring necessary.

This board can also be used with the examples from the Metro Experimenter's Guide ( ).

A good choice if you want something in the classic Arduino UNO form factor for use with the many available Arduino shields. The board is very powerful and runs CircuitPython very nicely

Itsy Bitsy M4 Starter Kit

A more hands on approach with some light soldering required.



CircuitPython Starter Kit with Adafruit Itsy Bitsy M4

You've heard about CircuitPython and maybe you want to get started fast with a breadboard-friendly microcontroller board! We recommend the ItsyBitsy M4 - a super fast chip...

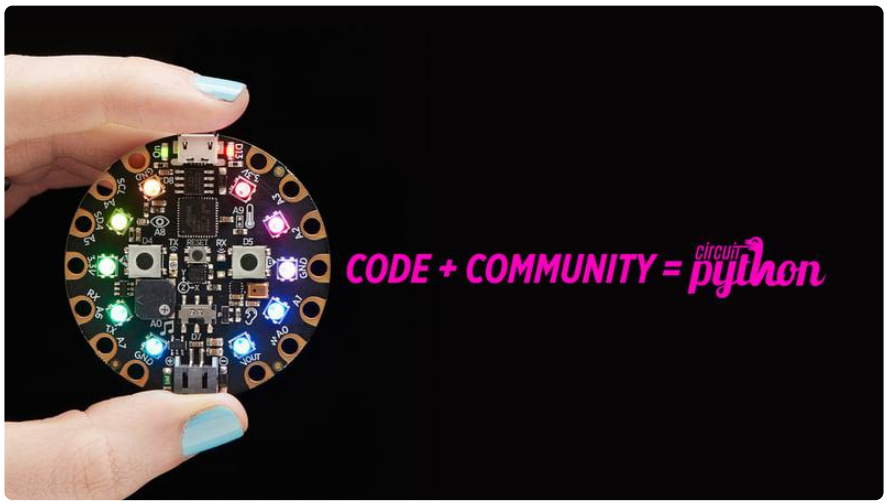
https://www.adafruit.com/product/4028

This kit does require some soldering to attach the header pins to the Itsy Bitsy M4. It also requires building up each circuit to use the various components. However, this does allow you to gain more experience by being more hands on with the hardware.

This a good choice if you are comfortable soldering and building circuits and want a more hands on experience. The board is very powerful and runs CircuitPython very nicely

---

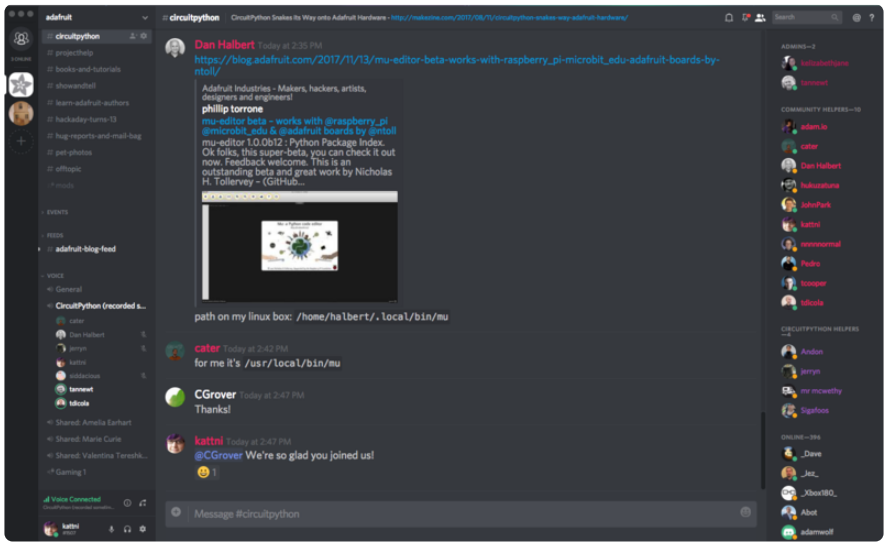
Welcome to the Community!



CircuitPython is a programming language that's super simple to get started with and great for learning. It runs on microcontrollers and works out of the box. You can plug it in and get started with any text editor. The best part? CircuitPython comes with an amazing, supportive community.

Everyone is welcome! CircuitPython is Open Source. This means it's available for anyone to use, edit, copy and improve upon. This also means CircuitPython becomes better because of you being a part of it. Whether this is your first microcontroller board or you're a seasoned software engineer, you have something important to offer the Adafruit CircuitPython community. This page highlights some of the many ways you can be a part of it!

Adafruit Discord



The Adafruit Discord server is the best place to start. Discord is where the community comes together to volunteer and provide live support of all kinds. From general discussion to detailed problem solving, and everything in between, Discord is a digital maker space with makers from around the world.

There are many different channels so you can choose the one best suited to your needs. Each channel is shown on Discord as "#channelname". There's the #help-with-projects channel for assistance with your current project or help coming up with ideas for your next one. There's the #show-and-tell channel for showing off your newest creation. Don't be afraid to ask a question in any channel! If you're unsure, #general is a great place to start. If another channel is more likely to provide you with a better answer, someone will guide you.

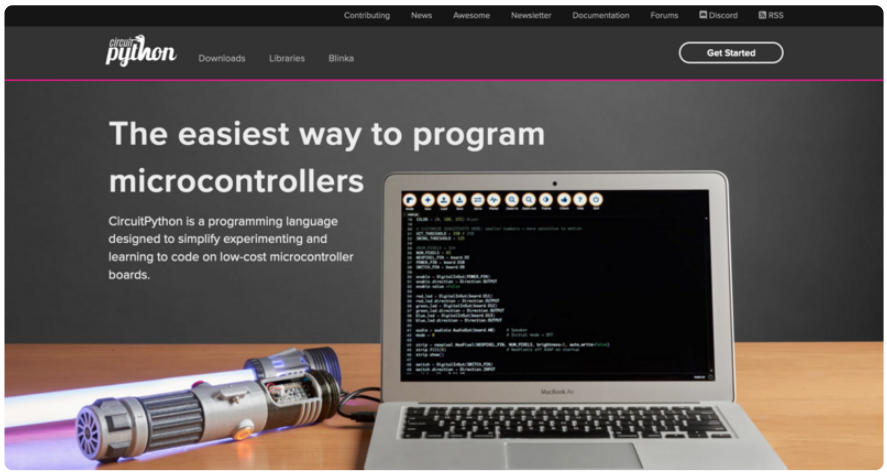
The help with CircuitPython channel is where to go with your CircuitPython questions. #help-with-circuitpython is there for new users and developers alike so feel free to ask a question or post a comment! Everyone of any experience level is welcome to join in on the conversation. Your contributions are important! The #circuitpython-dev channel is available for development discussions as well.

The easiest way to contribute to the community is to assist others on Discord. Supporting others doesn't always mean answering questions. Join in celebrating successes! Celebrate your mistakes! Sometimes just hearing that someone else has gone through a similar struggle can be enough to keep a maker moving forward.

The Adafruit Discord is the 24x7x365 hackerspace that you can bring your granddaughter to.

Visit https://adafru.it/discord ()to sign up for Discord. Everyone is looking forward to meeting you!

CircuitPython.org



Beyond the Adafruit Learn System, which you are viewing right now, the best place to find information about CircuitPython is circuitpython.org (). Everything you need to get started with your new microcontroller and beyond is available. You can do things like download CircuitPython for your microcontroller () or download the latest CircuitPython Library bundle (), or check out which single board computers support Blinka (). You can also get to various other CircuitPython related things like Awesome CircuitPython or the Python for Microcontrollers newsletter. This is all incredibly useful, but it isn't necessarily community related. So why is it included here? The Contributing page ().

Contributing

If you'd like to contribute to the CircuitPython project, the CircuitPython libraries are a great way to begin. This page is updated with daily status information from the CircuitPython libraries, including open pull requests, open issues and library infrastructure issues.

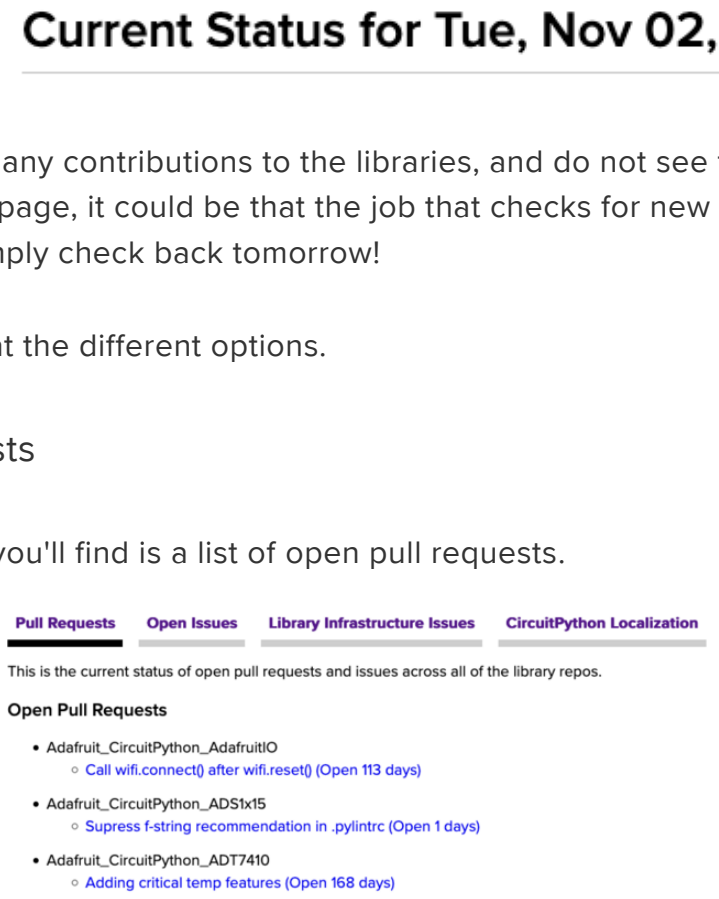
Do you write a language other than English? Another great way to contribute to the project is to contribute new localizations (translations) of CircuitPython, or update current localizations, using Weblate.

If this is your first time contributing, or you'd like to see our recommended contribution workflow, we have a guide on Contributing to CircuitPython with Git and Github. You can also find us in the #circuitpython channel on the Adafruit Discord.

Have an idea for a new driver or library? File an issue on the CircuitPython repo!

CircuitPython itself is written in C. However, all of the Adafruit CircuitPython libraries are written in Python. If you're interested in contributing to CircuitPython on the Python side of things, check out circuitpython.org/contributing (). You'll find information pertaining to every Adafruit CircuitPython library GitHub repository, giving you the opportunity to join the community by finding a contributing option that works for you.

Note the date on the page next to Current Status for:



If you submit any contributions to the libraries, and do not see them reflected on the Contributing page, it could be that the job that checks for new updates hasn't yet run for today. Simply check back tomorrow!

Now, a look at the different options.

Pull Requests

The first tab you'll find is a list of open pull requests.

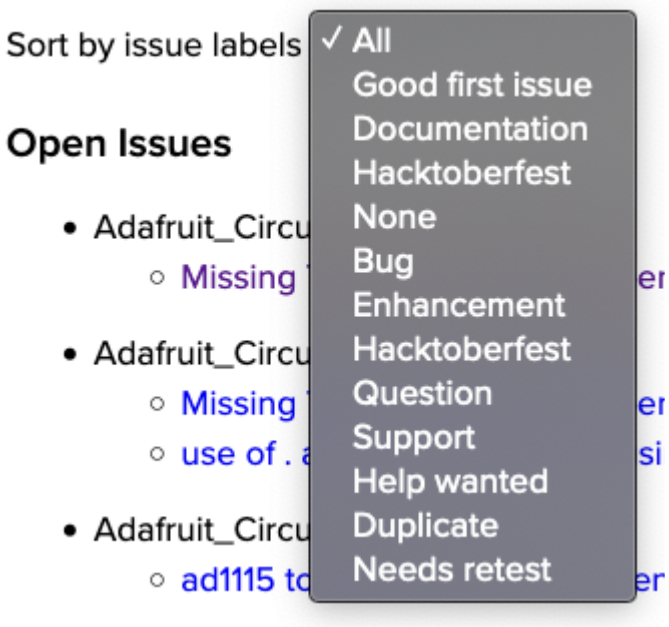
GitHub pull requests, or PRs, are opened when folks have added something to an Adafruit CircuitPython library GitHub repo, and are asking for Adafruit to add, or merge, their changes into the main library code. For PRs to be merged, they must first be reviewed. Reviewing is a great way to contribute! Take a look at the list of open pull requests, and pick one that interests you. If you have the hardware, you can test code changes. If you don't, you can still check the code updates for syntax. In the case of documentation updates, you can verify the information, or check it for spelling and grammar. Once you've checked out the update, you can leave a comment letting us know that you took a look. Once you've done that for a while, and you're more comfortable with it, you can consider joining the CircuitPythonLibrarians review team. The more reviewers we have, the more authors we can support. Reviewing is a crucial part of an open source ecosystem, CircuitPython included.

Open Issues

The second tab you'll find is a list of open issues.

GitHub issues are filed for a number of reasons, including when there is a bug in the library or example code, or when someone wants to make a feature request. Issues are a great way to find an opportunity to contribute directly to the libraries by updating code or documentation. If you're interested in contributing code or documentation, take a look at the open issues and find one that interests you.

If you're not sure where to start, you can search the issues by label. Labels are applied to issues to make the goal easier to identify at a first glance, or to indicate the difficulty level of the issue. Click on the dropdown next to "Sort by issue labels" to see the list of available labels, and click on one to choose it.



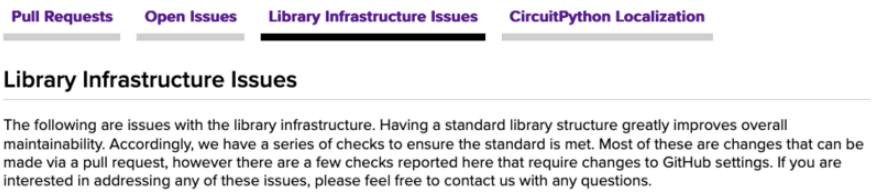
If you're new to everything, new to contributing to open source, or new to contributing to the CircuitPython project, you can choose "Good first issue". Issues with that label are well defined, with a finite scope, and are intended to be easy for someone new to figure out.

If you're looking for something a little more complicated, consider "Bug" or "Enhancement". The Bug label is applied to issues that pertain to problems or failures found in the library. The Enhancement label is applied to feature requests.

Don't let the process intimidate you. If you're new to Git and GitHub, there is a guide () to walk you through the entire process. As well, there are always folks available on Discord () to answer questions.

Library Infrastructure Issues

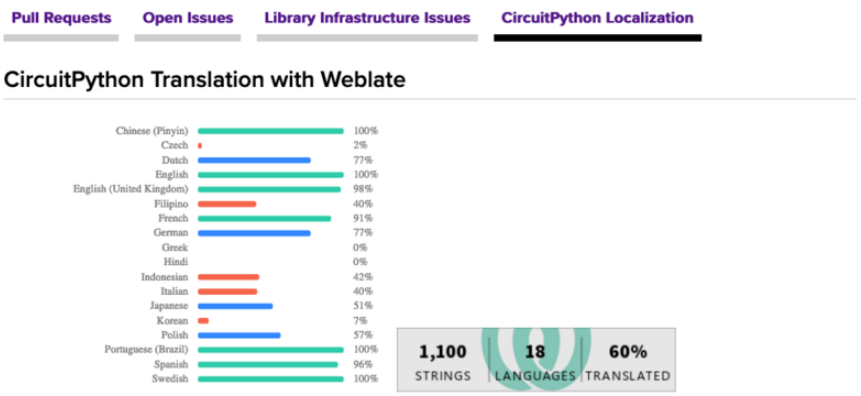
The third tab you'll find is a list of library infrastructure issues.



This section is generated by a script that runs checks on the libraries, and then reports back where there may be issues. It is made up of a list of subsections each containing links to the repositories that are experiencing that particular issue. This page is available mostly for internal use, but you may find some opportunities to contribute on this page. If there's an issue listed that sounds like something you could help with, mention it on Discord, or file an issue on GitHub indicating you're working to resolve that issue. Others can reply either way to let you know what the scope of it might be, and help you resolve it if necessary.

CircuitPython Localization

The fourth tab you'll find is the CircuitPython Localization tab.

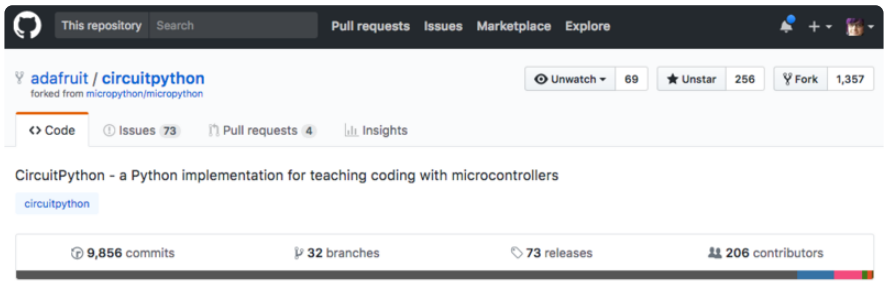


If you speak another language, you can help translate CircuitPython! The translations apply to informational and error messages that are within the CircuitPython core. It means that folks who do not speak English have the opportunity to have these messages shown to them in their own language when using CircuitPython. This is incredibly important to provide the best experience possible for all users.

CircuitPython uses Weblate to translate, which makes it much simpler to contribute translations. You will still need to know some CircuitPython-specific practices and a few basics about coding strings, but as with any CircuitPython contributions, folks are there to help.

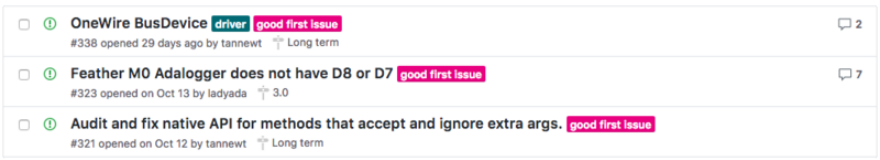
Regardless of your skill level, or how you want to contribute to the CircuitPython project, there is an opportunity available. The Contributing page () is an excellent place to start!

Adafruit GitHub



Whether you're just beginning or are life-long programmer who would like to contribute, there are ways for everyone to be a part of the CircuitPython project. The CircuitPython core is written in C. The libraries are written in Python. GitHub is the best source of ways to contribute to the CircuitPython core (), and the CircuitPython libraries (). If you need an account, visit https://github.com/ () and sign up.

If you're new to GitHub or programming in general, there are great opportunities for you. For the CircuitPython core, head over to the CircuitPython repository on GitHub, click on "Issues ()", and you'll find a list that includes issues labeled "good first issue ()" . For the libraries, head over to the Contributing page Issues list (), and use the drop down menu to search for "good first issue ()". These issues are things that have been identified as something that someone with any level of experience can help with. These issues include options like updating documentation, providing feedback, and fixing simple bugs. If you need help getting started with GitHub, there is an excellent guide on Contributing to CircuitPython with Git and GitHub ().



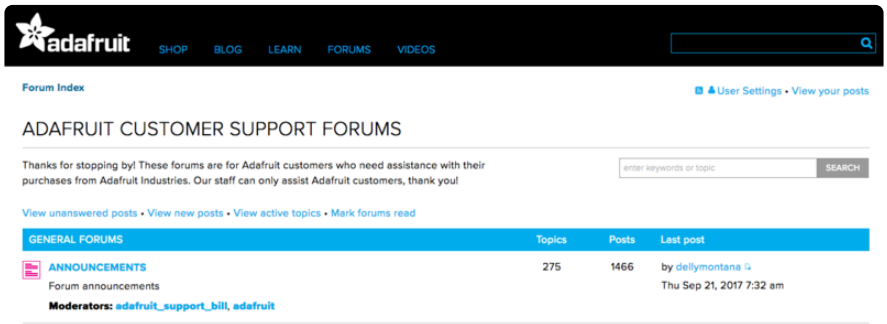
Already experienced and looking for a challenge? Checkout the rest of either issues list and you'll find plenty of ways to contribute. You'll find all sorts of things, from new driver requests, to library bugs, to core module updates. There's plenty of opportunities for everyone at any level!

When working with or using CircuitPython or the CircuitPython libraries, you may find problems. If you find a bug, that's great! The team loves bugs! Posting a detailed issue to GitHub is an invaluable way to contribute to improving CircuitPython. For CircuitPython itself, file an issue here (). For the libraries, file an issue on the specific library repository on GitHub. Be sure to include the steps to replicate the issue as well as any other information you think is relevant. The more detail, the better!

Testing new software is easy and incredibly helpful. Simply load the newest version of CircuitPython or a library onto your CircuitPython hardware, and use it. Let us know about any problems you find by posting a new issue to GitHub. Software testing on both stable and unstable releases is a very important part of contributing CircuitPython. The developers can't possibly find all the problems themselves! They need your help to make CircuitPython even better.

On GitHub, you can submit feature requests, provide feedback, report problems and much more. If you have questions, remember that Discord and the Forums are both there for help!

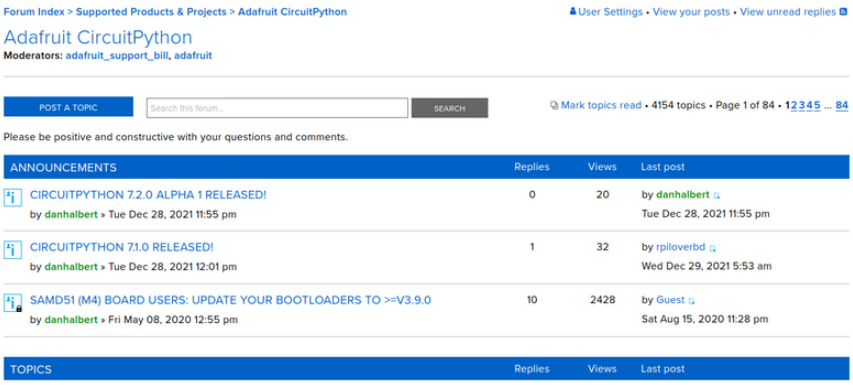
Adafruit Forums



The Adafruit Forums () are the perfect place for support. Adafruit has wonderful paid support folks to answer any questions you may have. Whether your hardware is giving you issues or your code doesn't seem to be working, the forums are always there for you to ask. You need an Adafruit account to post to the forums. You can use the same account you use to order from Adafruit.

While Discord may provide you with quicker responses than the forums, the forums are a more reliable source of information. If you want to be certain you're getting an Adafruit-supported answer, the forums are the best place to be.

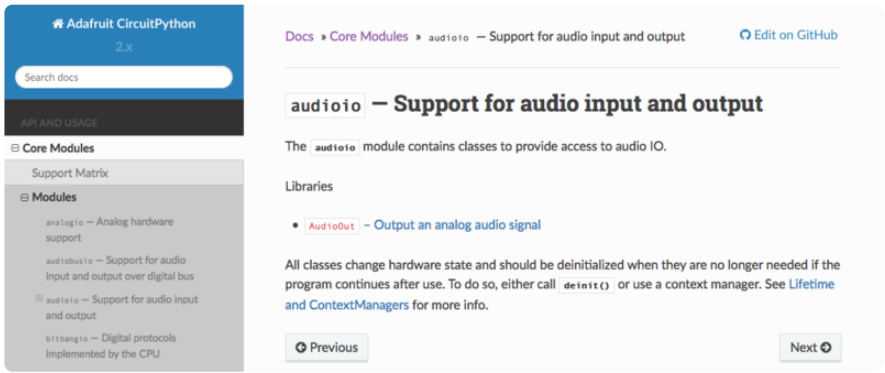
There are forum categories that cover all kinds of topics, including everything Adafruit. The Adafruit CircuitPython () category under "Supported Products & Projects" is the best place to post your CircuitPython questions.



Be sure to include the steps you took to get to where you are. If it involves wiring, post a picture! If your code is giving you trouble, include your code in your post! These are great ways to make sure that there's enough information to help you with your issue.

You might think you're just getting started, but you definitely know something that someone else doesn't. The great thing about the forums is that you can help others too! Everyone is welcome and encouraged to provide constructive feedback to any of the posted questions. This is an excellent way to contribute to the community and share your knowledge!

Read the Docs



Read the Docs () is a an excellent resource for a more detailed look at the CircuitPython core and the CircuitPython libraries. This is where you'll find things like API documentation and example code. For an in depth look at viewing and understanding Read the Docs, check out the CircuitPython Documentation () page!

Here is blinky:

```
import time
import digitalio
import board

led = digitalio.DigitalInOut(board.LED)
led.direction = digitalio.Direction.OUTPUT
while True:
    led.value = True
    time.sleep(0.1)
    led.value = False
    time.sleep(0.1)
```

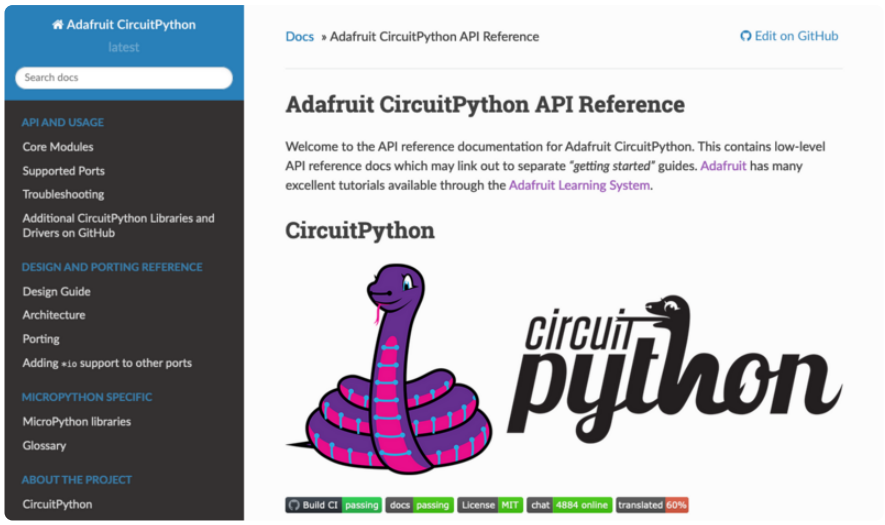
CircuitPython Documentation

You've learned about the CircuitPython built-in modules and external libraries. You know that you can find the modules in CircuitPython, and the libraries in the Library Bundles. There are guides available that explain the basics of many of the modules and libraries. However, there's sometimes more capabilities than are necessarily showcased in the guides, and often more to learn about a module or library. So, where can you find more detailed information? That's when you want to look at the API documentation.

The entire CircuitPython project comes with extensive documentation available on Read the Docs. This includes both the CircuitPython core () and the Adafruit CircuitPython libraries ().

CircuitPython Core Documentation

The CircuitPython core documentation () covers many of the details you might want to know about the CircuitPython core and related topics. It includes API and usage info, a design guide and information about porting CircuitPython to new boards, MicroPython info with relation to CircuitPython, and general information about the project.



The main page covers the basics including where to download CircuitPython, how to contribute, differences from MicroPython, information about the project structure, and a full table of contents for the rest of the documentation.

The list along the left side leads to more information about specific topics.

The first section is API and Usage. This is where you can find information about how to use individual built-in core modules, such as `time` and `digitalio`, details about the supported ports, suggestions for troubleshooting, and basic info and links to the li brary bundles. The Core Modules section also includes the Support Matrix, which is a table of which core modules are available on which boards.

The second section is Design and Porting Reference. It includes a design guide, archit ecture information, details on porting, and adding module support to other ports.

The third section is MicroPython Specific. It includes information on MicroPython and related libraries, and a glossary of terms.

The fourth and final section is About the Project. It includes further information including details on building, testing, and debugging CircuitPython, along with various other useful links including the Adafruit Community Code of Conduct.

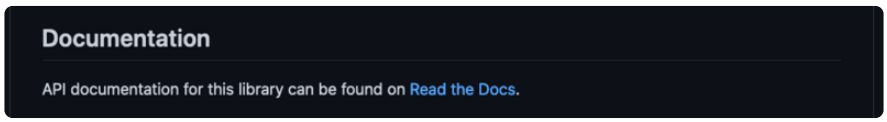
Whether you're a seasoned pro or new to electronics and programming, you'll find a wealth of information to help you along your CircuitPython journey in the documentation!

CircuitPython Library Documentation

The Adafruit CircuitPython libraries are documented in a very similar fashion. Each library has its own page on Read the Docs. There is a comprehensive list available her e (). Otherwise, to view the documentation for a specific library, you can visit the GitHub repository for the library, and find the link in the README.

For the purposes of this page, the LED Animation library () documentation will be featured. There are two links to the documentation in each library GitHub repo. The first one is the docs badge near the top of the README.

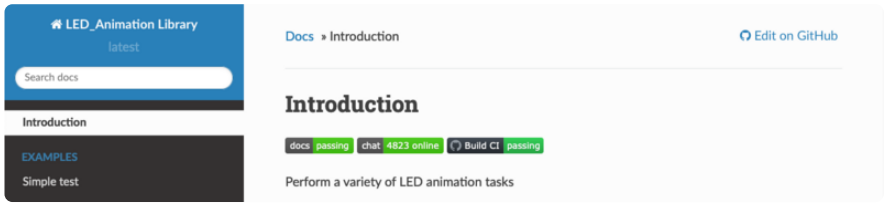
The second place is the Documentation section of the README. Scroll down to find it, and click on Read the Docs to get to the documentation.



Now that you know how to find it, it's time to take a look at what to expect.

Not all library documentation will look exactly the same, but this will give you some idea of what to expect from library docs.

The Introduction page is generated from the README, so it includes all the same info, such as PyPI installation instructions, a quick demo, and some build details. It also includes a full table of contents for the rest of the documentation (which is not part of the GitHub README). The page should look something like the following.



The left side contains links to the rest of the documentation, divided into three separate sections: Examples, API Reference, and Other Links.

Examples

The Examples section () is a list of library examples. This list contains anywhere from a small selection to the full list of the examples available for the library.

This section will always contain at least one example - the simple test example.

The simple test example is usually a basic example designed to show your setup is working. It may require other libraries to run. Keep in mind, it's simple - it won't showcase a comprehensive use of all the library features.

The LED Animation simple test demonstrates the Blink animation.

Simple test

Ensure your device works with this simple test.

examples/led_animation_simpletest.py

```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT

"""
This simpletest example displays the Blink animation.

For NeoPixel FeatherWing. Update pixel_pin and pixel_num to match your wiring if using
a different form of NeoPixels.
"""
import board
import neopixel
from adafruit_led_animation.animation.blink import Blink
from adafruit_led_animation.color import RED

# Update to match the pin connected to your NeoPixels
pixel_pin = board.D6
# Update to match the number of NeoPixels you have connected
pixel_num = 32

pixels = neopixel.NeoPixel(pixel_pin, pixel_num, brightness=0.5, auto_write=False)

blink = Blink(pixels, speed=0.5, color=RED)

while True:
    blink.animate()
```

In some cases, you'll find a longer list, that may include examples that explore other features in the library. The LED Animation documentation includes a series of examples, all of which are available in the library. These examples include demonstrations of both basic and more complex features. Simply click on the example that interests you to view the associated code.

When there are multiple links in the Examples section, all of the example content is, in actuality, on the same page. Each link after the first is an anchor link to the specified section of the page. Therefore, you can also view all the available examples by scrolling down the page.

You can view the rest of the examples by clicking through the list or scrolling down the page. These examples are fully working code. Which is to say, while they may rely on other libraries as well as the library for which you are viewing the documentation, they should not require modification to otherwise work.

API Reference

The API Reference section () includes a list of the library functions and classes. The API (Application Programming Interface) of a library is the set of functions and classes the library provides. Essentially, the API defines how your program interfaces with the functions and classes that you call in your code to use the library.

There is always at least one list item included. Libraries for which the code is included in a single Python (.py) file, will only have one item. Libraries for which the code is multiple Python files in a directory (called subpackages) will have multiple items in this list. The LED Animation library has a series of subpackages, and therefore, multiple items in this list.

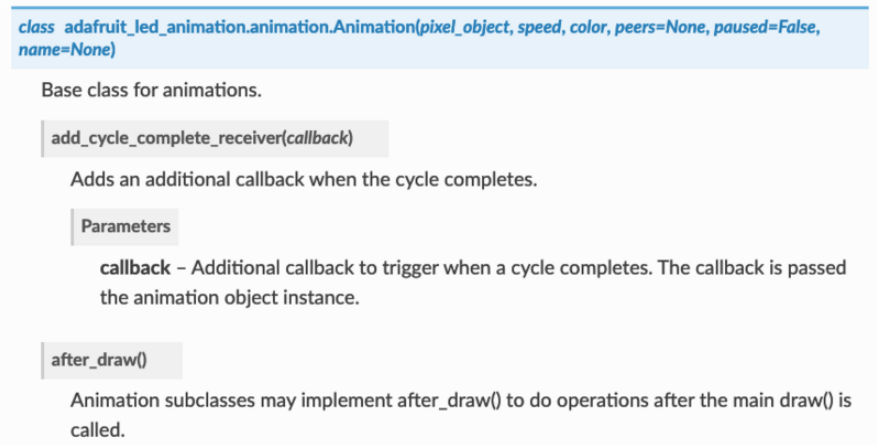
Click on the first item in the list to begin viewing the API Reference section.

As with the Examples section, all of the API Reference content is on a single page, and the links under API Reference are anchor links to the specified section of the page.

When you click on an item in the API Reference section, you'll find details about the classes and functions in the library. In the case of only one item in this section, all the

available functionality of the library will be contained within that first and only subsection. However, in the case of a library that has subpackages, each item will contain the features of the particular subpackage indicated by the link. The documentation will cover all of the available functions of the library, including more complex ones that may not interest you.

The first list item is the animation subpackage. If you scroll down, you'll begin to see the available features of animation. They are listed alphabetically. Each of these things can be called in your code. It includes the name and a description of the specific function you would call, and if any parameters are necessary, lists those with a description as well.



class adafruit_led_animation.animation.Animation(*pixel_object*, *speed*, *color*, *peers=None*, *paused=False*, *name=None*)

Base class for animations.

add_cycle_complete_receiver(*callback*)

Adds an additional callback when the cycle completes.

Parameters

callback – Additional callback to trigger when a cycle completes. The callback is passed the animation object instance.

after_draw()

Animation subclasses may implement after_draw() to do operations after the main draw() is called.

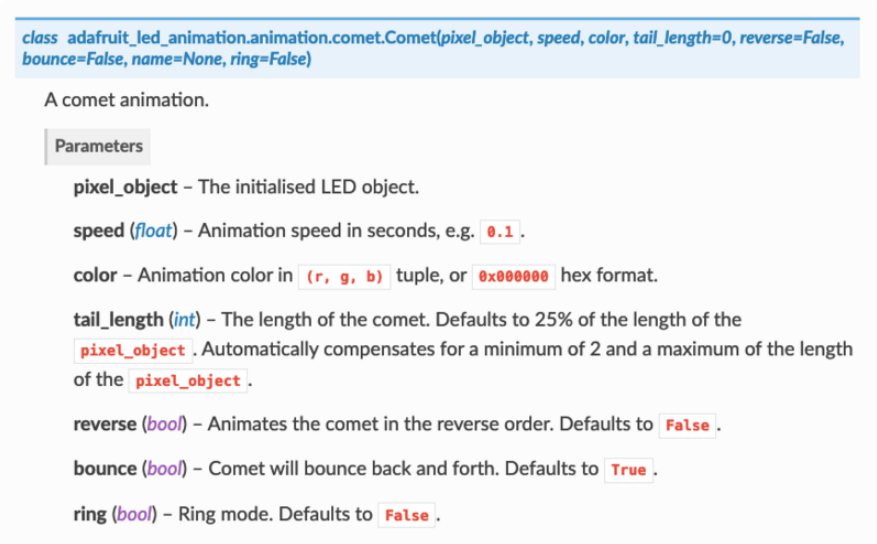
You can view the other subpackages by clicking the link on the left or scrolling down the page. You may be interested in something a little more practical. Here is an example. To use the LED Animation library Comet animation, you would run the following example.

```
# SPDX-FileCopyrightText: 2021 Kattni Rembor for Adafruit Industries
# SPDX-License-Identifier: MIT

"""
This example animates a jade comet that bounces from end to end of the strip.

For QT Py Haxpress and a NeoPixel strip. Update pixel_pin and pixel_num to match
your wiring if
using a different board or form of NeoPixels.

This example will run on SAMD21 (M0) Express boards (such as Circuit Playground
Express or QT Py
Haxpress), but not on SAMD21 non-Express boards (such as QT Py or Trinket).
"""
import board
import neopixel

from adafruit_led_animation.animation.comet import Comet
from adafruit_led_animation.color import JADE

# Update to match the pin connected to your NeoPixels
pixel_pin = board.A3
# Update to match the number of NeoPixels you have connected
pixel_num = 30
```

```
pixels = neopixel.NeoPixel(pixel_pin, pixel_num, brightness=0.5, auto_write=False)

comet = Comet(pixels, speed=0.02, color=JADE, tail_length=10, bounce=True)

while True:
    comet.animate()
```

Note the line where you create the `comet` object. There are a number of items inside the parentheses. In this case, you're provided with a fully working example. But what if you want to change how the comet works? The code alone does not explain what the options mean.

So, in the API Reference documentation list, click the `adafruit_led_animation.animation.comet` link and scroll down a bit until you see the following.



class adafruit_led_animation.animation.comet.Comet(*pixel_object*, *speed*, *color*, *tail_length=0*, *reverse=False*, *bounce=False*, *name=None*, *ring=False*)

A comet animation.

Parameters

pixel_object – The initialised LED object.

speed (*float*) – Animation speed in seconds, e.g. `0.1`.

color – Animation color in `(r, g, b)` tuple, or `0x000000` hex format.

tail_length (*int*) – The length of the comet. Defaults to 25% of the length of the `pixel_object`. Automatically compensates for a minimum of 2 and a maximum of the length of the `pixel_object`.

reverse (*bool*) – Animates the comet in the reverse order. Defaults to `False`.

bounce (*bool*) – Comet will bounce back and forth. Defaults to `True`.

ring (*bool*) – Ring mode. Defaults to `False`.

Look familiar? It is! This is the documentation for setting up the comet object. It explains what each argument provided in the comet setup in the code meant, as well as the other available features. For example, the code includes `speed=0.02`. The documentation clarifies that this is the "Animation speed in seconds". The code doesn't include `ring`. The documentation indicates this is an available setting that enables "Ring mode".

This type of information is available for any function you would set up in your code. If you need clarification on something, wonder whether there's more options available, or are simply interested in the details involved in the code you're writing, check out the documentation for the CircuitPython libraries!

Other Links

This section is the same for every library. It includes a list of links to external sites, which you can visit for more information about the CircuitPython Project and Adafruit.

That covers the CircuitPython library documentation! When you are ready to go beyond the basic library features covered in a guide, or you're interested in understanding those features better, the library documentation on Read the Docs has you covered!

Advanced Setup

CircuitPython is designed to lower the barrier for entry to programming and electronics, which makes it excellent for beginners. It's also extremely extensible to far more complex projects, meaning it's also great for folks with more experience. If you have been programming for a while, or you're ready to explore some of the more advanced options available for working with CircuitPython, this section is for you!

Recommended Editors

The CircuitPython code on your board detects when the files are changed or written and will automatically re-start your code. This makes coding very fast because you save, and it re-runs.

However, you must wait until the file is done being saved before unplugging or resetting your board! On Windows using some editors this can sometimes take up to 90 seconds, on Linux it can take 30 seconds to complete because the text editor does not save the file completely. Mac OS does not seem to have this delay, which is nice!

This is really important to be aware of. If you unplug or reset the board before your computer finishes writing the file to your board, you can corrupt the drive. If this happens, you may lose the code you've written, so it's important to backup your code to your computer regularly.

To avoid the likelihood of filesystem corruption, use an editor that writes out the file completely when you save it. Check out the list of recommended editors below.

Recommended editors

- mu () is an editor that safely writes all changes (it's also our recommended editor!)
- emacs () is also an editor that will fully write files on save ()
- Sublime Text () safely writes all changes
- Visual Studio Code () appears to safely write all changes
- gedit on Linux appears to safely write all changes
- IDLE (), in Python 3.8.1 or later, was fixed () to write all changes immediately
- Thonny () fully writes files on save

Recommended only with particular settings or add-ons

- vim () / vi safely writes all changes. But set up vim to not write swapfiles () (.swp files: temporary records of your edits) to CIRCUITPY. Run vim with `vim -n`, set the `no swapfile` option, or set the `directory` option to write swapfiles elsewhere. Otherwise the swapfile writes trigger restarts of your program.
- The PyCharm IDE () is safe if "Safe Write" is turned on in Settings->System Settings->Synchronization (true by default).
- If you are using Atom (), install the fsync-on-save package () or the language-circuitpython package () so that it will always write out all changes to files on CIR CUITPY.
- SlickEdit () works only if you add a macro to flush the disk ().

The editors listed below are specifically NOT recommended!

Editors that are NOT recommended

- notepad (the default Windows editor) and Notepad++ can be slow to write, so the editors above are recommended! If you are using notepad, be sure to eject the drive.
- IDLE in Python 3.8.0 or earlier does not force out changes immediately.

- nano (on Linux) does not force out changes.
- geany (on Linux) does not force out changes.
- Anything else - Other editors have not been tested so please use a recommended one!

Library File Types and Frozen Libraries

This guide has already covered the basics of CircuitPython library usage. Copy a library to your CIRCUITPY drive, and it's available for use in your code.

However, you may have noticed there are two bundles available for download, or that on some boards, you don't have to copy a library over to be able to use it. What gives? This page covers the deeper details about library files, how to use them, why to use one over the other, and much more.

Library files are available in two different file formats: .mpy and .py. There are separate library bundles for each file type. Libraries are accessible to CircuitPython in three different ways: .mpy files, .py files, or frozen modules. The next few sections detail the differences between the file types, and the various ways to access libraries.

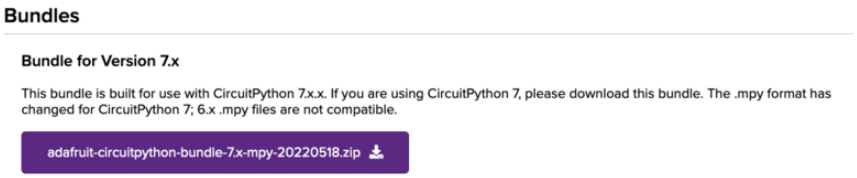
RAM vs Filesystem Space

This page refers regularly to library file memory (RAM) usage. "Memory" on this page means RAM (Random Access Memory), which is the read/write memory in the microcontroller. RAM and memory are used interchangeably throughout this page.

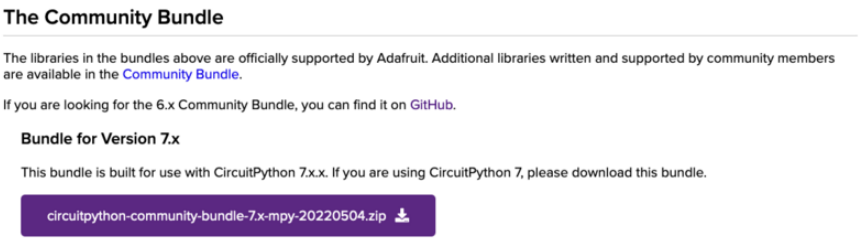
There is one section that references filesystem space. Filesystem space is not RAM; it is where files are stored.

.mpy Library Files

The most commonly used library file type is an .mpy file. You can download the .mpy version of a library directly from its GitHub repository, but the simplest, most convenient way to get them is by downloading the appropriate bundle from the Libraries page () on circuitpython.org (). The first bundle on the Libraries page is the .mpy bundle. This bundle provides every library in the Adafruit CircuitPython Bundle in the .mpy format.



The Community Bundle, which is made up of community sourced and maintained libraries, is also available in an .mpy format, further down the Libraries page.



What is an .mpy file?

An .mpy file is a file that has been generated by running the `mpy-cross` tool on a .py file, which compiles the .py file to bytecode. This process removes the doc strings and does some minor optimisations. Further, comments, whitespace, and type hints don't translate into bytecode, so they are also not present following compilation. The result is a file that is smaller and uses less RAM memory. `mpy-cross` makes it possible to import some libraries that couldn't otherwise be imported as .py files on smaller boards because they take up too much RAM to compile on the board.

Basically, you can provide CircuitPython with an .mpy file, it is loaded into RAM, but as it is already compiled, it does not need to be compiled on the fly. Between that and the minor optimisation that comes with the .mpy format, it uses less memory than a .py file. One caveat is that .mpy files are not human-readable. If you want to look at the code directly, you'll need to use a .py file.

To use an .mpy formatted library, you simply copy over the .mpy library file to to the /lib folder on your CIRCUITPY drive. For example, if you're using a demo that requires the NeoPixel library, you would copy the neopixel.mpy file to your /lib folder.

If you're interested in modifying the library, you'll find that you can't edit an .mpy file. For that, you need to switch to a .py file.

Creating an .mpy File

Creating your own .mpy files out of .py files is a quick and simple process using `mpy-cross`.

Download the latest `mpy-cross` for your operating system from here (), ensuring you choose the version of mpy-cross that matches your version of CircuitPython. Builds are available for Windows, macOS, x64 Linux, and Raspberry Pi Linux. For example, if you're using CircuitPython 7.3.0 with MacOS Big Sur, you could download mpy-cross-macos-bigsur-7.3.0-arm64. For ease of use, rename the file you download to mpy-cross.

To make an .mpy file, you will first need a .py file to work with. Open a terminal program or commandline, `cd` into the directory containing mpy-cross, and run the following:

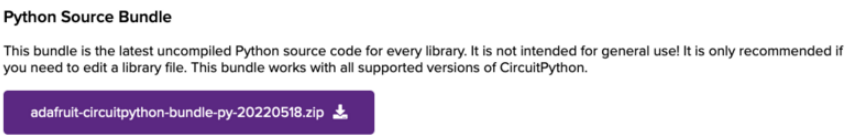
```
./mpy-cross path/to/your-library-file.py
```

This will create your-library-file.mpy in the same directory as the original .py file.

Then, copy your-library-file.mpy to your CIRCUITPY drive to use it. Easy as that!

.py Library Files.

Every CircuitPython library is available as a .py file. You can download the .py library files individually from their respective GitHub repositories. The quicker, more convenient way is to download the .py version of the Adafruit CircuitPython Bundle from the Libraries page. It is available as the "Python Source Bundle".



The Community Bundle, which is made up of community sourced and maintained libraries, is available in a .py format, further down the Libraries page. It is also titled "Python Source Bundle".

What is a .py file?

A .py file is a Python file that you can open and modify in your favorite Python editor. However, unless you're planning to modify a library and test your modifications, it is, in general, better to stick with .mpy files. Basically, .py files require more memory, and trying to use them on smaller microcontrollers like the SAMD21 (M0) can result in memory allocation failures when the library is imported in your code.

Frozen Libraries

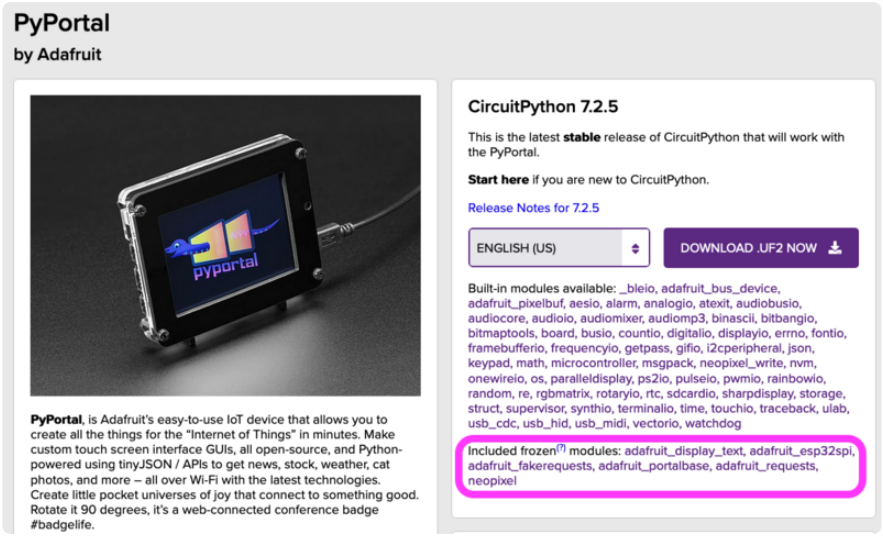
In some cases, library modules are "frozen" into the CircuitPython build for a specific board. You won't find a "frozen" bundle anywhere - there is nowhere to download frozen libraries, because they are built into CircuitPython for specific boards (i.e. only builds for boards that require frozen libraries). A library being frozen into CircuitPython means you can access that library without copying any files to CIRCUITPY/lib.

What is a frozen library?

Some microcontroller boards, such as the Circuit Playground Express, simply do not have enough memory to run code from some library files, even if they are .mpy format. In these cases, the applicable libraries and possibly the dependencies are included in the CircuitPython build for that specific board. Frozen libraries are any libraries that are included in CircuitPython builds for various boards. There are two advantages to doing this: it ensures code runs if there are otherwise memory issues, but more practically speaking, it means you can access these libraries in your code without copying the files to the CIRCUITPY/lib directory.

An excellent example of a board requiring frozen modules is the Circuit Playground Express. The CPX is easiest to use with the Adafruit CircuitPlayground Library. Many of the guides in the Adafruit Learn System for the CPX use the CircuitPlayground library. The problem is, if you copy the CircuitPlayground library, and its dependencies (Adafruit CircuitPython HID, LIS3DH, Thermistor, and NeoPixel) to the CPX, your code will quickly fail to run due to insufficient available memory. The CircuitPlayground Library imports all of the dependencies, so when you import it in your code, you're importing the rest of the list as well. So, what can you do? Freeze all of these libraries into the CPX CircuitPython build!

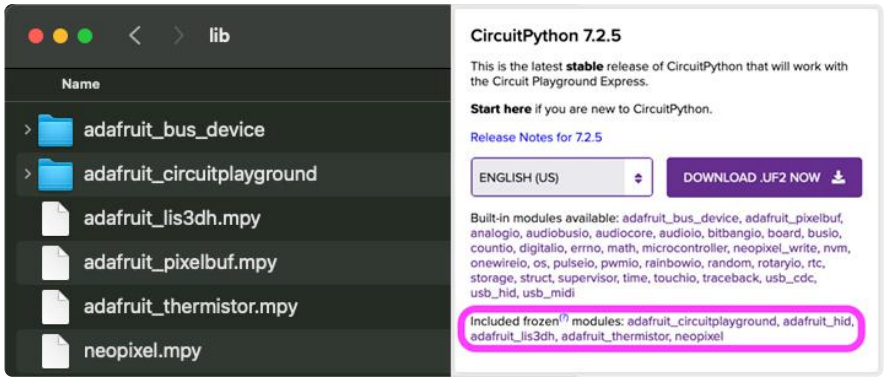
If you're wondering whether your microcontroller board has any frozen libraries in its CircuitPython build, you can check out the Downloads page on circuitpython.org () for your specific board. Search for your board, and click on it to open its specific download page. Under each version of CircuitPython currently available for your board, you'll find a list of "Built-in modules available:", and, in the event that your build contains frozen libraries, a list of "Included frozen modules:", as well. The frozen modules list includes all libraries frozen into CircuitPython for your board. The following shows the PyPortal on circuitpython.org (), including its list of frozen modules, highlighted in magenta in the bottom right corner of the image below.



Project Bundle and Frozen Libraries

Project Bundle basics () are covered on the CircuitPython Libraries page. However, it's important to understand how the Project Bundle works with frozen libraries. The Project Bundle isn't aware of the frozen libraries. Therefore, the Download Project Bundle button downloads a zip containing all of the library files necessary for a given example, whether or not the libraries are frozen into CircuitPython for a particular board build. This won't cause issues with running the libraries (continue reading this page for details), but it does mean that the libraries are taking up filesystem space. If this causes issues for you, you can delete any library files from the /lib directory that are frozen into the CircuitPython build for your board.

Here is an example using the Circuit Playground Express. Below are the /lib directory from the the Project bundle in the Piano in the Key of Lime guide (), and the list of frozen modules in CircuitPython for CPX as available on the Downloads page ().



Note that adafruit_bus_device is a special case. It is built into CircuitPython for most boards as an actual module (as you can see in the "Built-in modules available:" list). The Project Bundle doesn't know about that either, so it includes

it in the download. As long as you see it on the Downloads page, you can remove it from your microcontroller as well.

As you can see, all of the "Included frozen modules:" are also in the /lib directory in downloaded from the Project Bundle. Therefore, if you begin to run out of filesystem space on your Circuit Playground Express, you can delete the files from the /lib direct ory that match the list on the Downloads page to make more available. This applies to a project bundle for any microcontroller that contains frozen libraries..

Library File Priority

CircuitPython looks for specific library file types in specific locations on a microcontroller, both in a particular order. You can use this to your advantage when trying to use updated or modified libraries.

What if you want to test modifications or updates you made to a library file that is on your board as an .mpy file, or a library that is frozen into CircuitPython? There are a couple of ways to do this. To test your updates, you can copy the updated .py library file to the CIRCUITPY drive in the proper location. Specific to only frozen libraries, you can create your own build of CircuitPython with the change frozen in.

What library files does CircuitPython look for?

Library development requires the ability to test library changes live on a microcontroller to ensure they work properly. This means making an update to a .py file, copying it to CIRCUITPY, and running an example that uses your update. However, once you get into the development cycle, you may end up with multiple copies of a library on your board in different formats. How can you be sure CircuitPython is running the updated file? Check the format.

CircuitPython will attempt to run a .py file first. This means if you have a .py library file and a .mpy file of the same library in the same directory, CircuitPython will run the .py.

For example, if you have neopixel.py and neopixel.mpy in your /lib folder on CIRCUITP Y, the neopixel.py file will take precedence over the neopixel.mpy file. Therefore, when you `import neopixel` in your code.py file to test your updates, it will import your updated library. The same results would occur if you had the same .py and .mpy file in root on your CIRCUITPY drive.

This can be used to your advantage, but it can also lead to potential issues. The Com mon Issues and Solutions section () below has more details.

The location of various library files on CIRCUITPY also plays a part in how CircuitPython decides which file to choose. Find out the specifics in the next section.

Where does CircuitPython look for library files?

Testing your library modifications on a microcontroller is crucial. Frozen libraries had the potential for making that more difficult. as you cannot remove them from the board. So, CircuitPython looks for library files in different locations in a specified order, in a way that allows you to "override" libraries in other locations. How do you know what that order is? That's where `sys.path` comes in.

In CircuitPython (and Python), `sys.path` is a variable in the `sys` module that returns a list of strings that specify the search path for modules or libraries. More simply put, it allows for telling CircuitPython exactly where to look, and in what order, for library files on the CIRCUITPY drive.

To find out the specifics, you can run the following two lines of code in code.py or from the REPL on the board you're currently working with. If there are libraries frozen into CircuitPython for the particular board, the results will look like this:

```
&gt;&gt;&gt; import sys
&gt;&gt;&gt; print(sys.path)
['', '/', '.frozen', '/lib']
```

If there are not frozen libraries, the results will look like this:

```
&gt;&gt;&gt; import sys
&gt;&gt;&gt; print(sys.path)
['', '/', '/lib']
```

What does this actually mean? You can break down the results as follows.

If there are frozen modules built into the build of CircuitPython for the board you're using, CircuitPython looks for library files in the following locations in this exact order:

- Library files in the current directory (CIRCUITPY/)
- Library files in root (CIRCUITPY/)
- Fozen modules (.frozen is a fake directory for this purpose, as the frozen modules are not in the filesystem)
- Library files in the library folder (CIRCUITPY/lib)

If there are no frozen modules built into the build of CircuitPython for the board you're using, CircuitPython looks for library files in the following locations in this exact order:

- Library files in the current directory (CIRCUITPY/)
- Library files in root (CIRCUITPY/)
- Library files in the library folder (CIRCUITPY/lib)

What does all of this mean practically speaking? Here are a couple of examples.

1. You put the .mpy version of a library into the /lib folder, and you find an issue. You obtain the .py version of the library file, make your changes, and are ready to test. Instead of deleting the .mpy in the /lib folder, you can copy your modified .py file to root on CIRCUITPY/, and it will "override" the .mpy as CircuitPython will find your modified .py first, and run it.
2. You are using a microcontroller board with a particular library frozen into CircuitPython, and you find an issue. You obtain the .py version of the library, make your changes, and are ready to test. You can place the modified .py file in root on CIRCUITPY/, and it will "override" the frozen module as CircuitPython will find your modified .py first, and run it.

You can also run `mpy-cross` on your modified .py file, and place the .mpy file in root on CIRCUITPY/, and you will "override" the frozen modules and everything in the lib/ folder. This is necessary if you find that using the .py file causes your code to fail due to memory issues.

Ostensibly all of this also means you can put libraries in root on your CIRCUITPY/ drive, and they will run as they do in the /lib folder. However, if you always copy library files to the /lib folder, it means this technique for testing is always available without needing to delete files before you test.

Building CircuitPython with an Updated Frozen Library

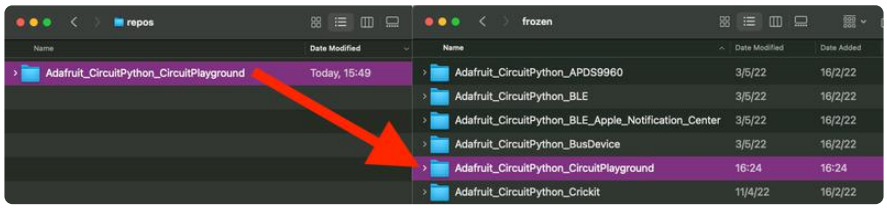
With some boards, for example the Circuit Playground Express, you'll find that overriding the frozen modules with a .py file will cause running your code to fail due to a memory allocation failure (). In some cases, even if you run `mpy-cross` on the .py file, your code will still fail. This means you may have to create your own build of CircuitPython with your modified library frozen into it.

The more difficult but most realistic way to test a frozen library modification is to build CircuitPython with your modified library included. It's the most realistic way to test because it ensures that the changes still fit into the specific CircuitPython build. It also

allows you to test a library when, even as an .mpy, the code fails to run due to memory constraints. However, for some folks this is an intimidating prospect when they wanted to test a simple change they made to a library.

This is why there are detailed instructions () for creating your own build of CircuitPython. Follow the instructions as is, up to the Build CircuitPython section (). Before you run the actual build, you'll want to create a new branch () in your local copy of the CircuitPython repository, and then copy the modified library into the froze n/ directory.

Before you can do that, you'll want to follow the steps here () to clone a copy of the library you're modifying to your computer. Make your modifications within the local library directory. When you're ready to test, you'll copy the entire library directory, e.g. the Adafruit_CircuitPython_CircuitPlayground/ directory, into the frozen/ directory in root of your local copy of the CircuitPython repository. This will replace the existing version with your updated version.



Once you've updated the library directory in frozen/ with your modified version of the library, you're ready to build. Continue following the instructions in the Building CircuitPython () guide to finish building your version of CircuitPython. Remember to change the board you're building for, if it is different from the one in the example. Once built, place your board into bootloader mode, and copy the firmware.uf2 file to the *BOOT drive.

Now, you're ready to test your changes!

What's the Real Difference?

You now understand the different ways libraries are supplied to CircuitPython, and which options work best for different use cases. But, how do you know that these separate options actually make a difference in memory usage? There's a simple way to see the actual difference, by the numbers, for yourself.

Memory Usage between .mpy and .py

There are two parts of importing a Python file that use extra memory. The code compilation process, and the resulting bytecode, which needs to be stored in memory during and after the compilation. Remember, .mpy files are already compiled, and do not need to be compiled on import.

Essentially, the import of an .mpy loads compiled code, and the import of a .py must compile the code on import. In both cases, the compiled code lives in RAM, and the resulting compiled code is basically the same size. However, the compilation process takes extra RAM temporarily during compilation, and if there's not enough RAM available, it will fail. Compilation may also cause some fragmentation, but this is mostly avoided in CircuitPython.

Think of it this way. You can bring a ready-to-eat cake to your friend's house, or you can show up with the plan to bake the same cake at your friend's house. Both completed cakes will take up the same amount of space. However, waiting to bake the cake means you'll need more space and resources before the cake is ready to eat, to combine the ingredients, bake it, and frost it. If it turns out your friend doesn't have the space available, or is missing a crucial ingredient, you wouldn't be able to complete the cake. Basically, if you're unsure whether you'll have everything needed to bake a cake at your friend's house, it's better to show up with a completed cake to ensure there is cake to be had!

If you're unsure whether your microcontroller board has the memory resources needed to run a .py version of a library, the simplest solution is to choose a .mpy file instead.

Using `gc.mem_free()` to Check Your Bytes

CircuitPython has a built-in `gc` (garbage collector) module, which includes `gc.mem_free()`. The `mem_free()` functionality is specific to CircuitPython (and MicroPython), and is therefore not available in the CPython `gc` module.

When you run `gc.mem_free()` in the REPL, or `print(gc.mem_free())` in code.py, it returns the number of bytes of available heap RAM. This is the amount of RAM left for importing further libraries, and executing code. Specifically for our purposes, though, it will tell us how much memory is used when importing the same library as a .py file, an .mpy file, and a frozen module.

For this example, you will use the NeoPixel library on the Circuit Playground Express. The NeoPixel library is frozen into the CircuitPython build for CPX, as well as being available in the two bundles as neopixel.mpy and neopixel.py. To prepare for this, download both the mpy bundle and the Python Source Bundle. To get started, you'll want to connect to the serial console and enter the REPL.

For each of these examples, you'll first run the following from the REPL:

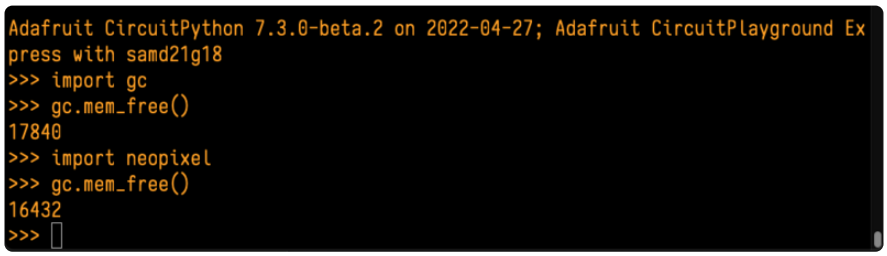
```
>>> import gc
>>> gc.mem_free()
```

This provides you with a baseline of available memory before importing any further libraries.

First up, you'll test out how much memory is used when importing the frozen version of the NeoPixel library. Make sure you have no libraries copied anywhere on your CIR CUITPY drive, including in the lib folder!

Once you've determined the baseline, as explained above, run the following from the REPL:

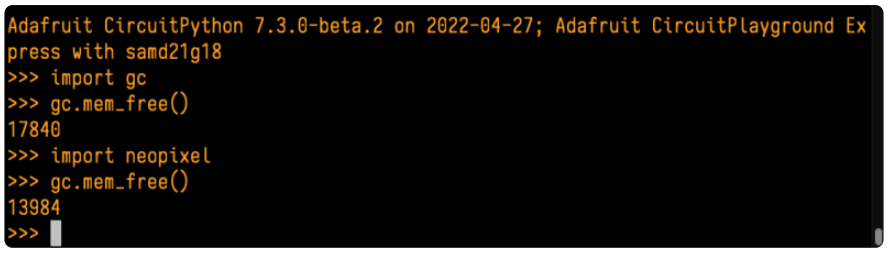
```
>>> import neopixel
>>> gc.mem_free()
```



As you can see, the baseline available memory is 17840 bytes. Once you import the frozen version of NeoPixel, you have 16432 bytes available. Importing the frozen version requires 1408 bytes.

Now, copy the neopixel.mpy file to the root of your CIRCUITPY/ drive. Remember, CircuitPython looks in the root directory for libraries before checking the frozen modules. This will ensure that you're importing the version you intend to.

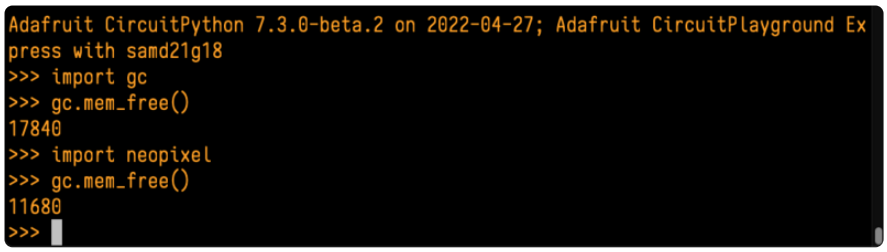
Follow the same steps as above to get a baseline and show the memory available following import.



The baseline is the same, however, the available memory following import is now 13984 bytes. Importing neopixel.mpy requires 3856 bytes.

Next, delete the neopixel.mpy file from the CIRCUITPY drive. Copy the neopixel.py file to the root of your CIRCUITPY/ drive.

Follow the same steps as above to get a baseline and show the memory available following import.



The baseline is the same, however, the available memory following import is now 11680 bytes. Importing neopixel.py requires 6160 bytes.

Importing each file type requires significantly different amounts of memory! The available memory following each file type import is what's left to import more libraries, and run your code. You can see how using different file types might impact the ability to use memory-constrained microcontroller boards, and run more complicated code.

Common Issues and Solutions

This page has explained the different ways you can obtain, access, and test modifications to CircuitPython libraries. Here are some issues you may run into in the course of things, and suggestions as to how to deal with them.

.mpy Failing When It Worked Previously

You are certain that you downloaded the .mpy version of the library, and that the code ran previously using the .mpy file, but now, it's failing due to memory usage. The most common reason for this that there is a leftover .py version of the same library in the same directory (e.g. the .py and .mpy are both in /lib), or the .py file is in an "override"

directory (e.g. the .py is in / and the .mpy is in /lib). In both of these examples, CircuitPython is going to load the .py file.

If you run into this, check out the contents of your CIRCUITPY drive, and ensure that you don't have any leftover .py files in / or in /lib. More often than not, you'll find that removing the .py file resolves this situation.

Memory Allocation

There are answers in the FAQ in this guide () as to what a memory error is, and how to avoid it if you're interested in more detail. What it boils down to is your microcontroller board running out of memory to import libraries or execute code.

The Circuit Playground Express is a memory-constrained microcontroller board. Substituting the CircuitPlayground library for the NeoPixel library in the byte-checking example results in similar behavior for the frozen module and the .mpy file. However, if you try to import the CircuitPlayground library as a .py package, it will fail to import with an error similar to the one below.

```
Adafruit CircuitPython 7.3.0-beta.2 on 2022-04-27; Adafruit CircuitPlayground Ex
press with samd21g18
>>> import gc
>>> gc.mem_free()
17840
>>> from adafruit_circuitplayground import cp
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "adafruit_circuitplayground/__init__.py", line 12, in <module>
  File "adafruit_circuitplayground/express.py", line 31, in <module>
MemoryError: memory allocation failed, allocating 1097 bytes
```

The CircuitPlayground library is an excellent example of this because the library is quite large, has a significant number of dependencies imported in it, and absolutely cannot be run on the CPX as a .py file. This is not the case for all libraries frozen into all builds of CircuitPython (as shown with NeoPixel), so you will not always get this result when attempting to test modifications or updates.

Memory Fragmentation

A memory allocation failure does not necessarily mean you are entirely out of memory. CircuitPython memory can become fragmented the more you do, and have smaller chunks of memory available across the entire heap that add up to the amount needed, but not a single chunk big enough to allocate for a particular task. Fragmentation can block allocation of any chunk of memory that needs to be contiguous, such as a large buffer. In the context of importing a library file, it

specifically means that there is not a single large enough chunk of memory to allocate the memory requested.

In the case of importing a .mpy file, the compiled bytecode is imported across multiple chunks. In the case of importing a .py file, the memory needed for compilation may exceed the size of the chunks available, in which case, you would run out of memory before there is even any bytecode to import.

CircuitPython does do some optimisation to avoid memory fragmentation. Once a library is imported successfully, it is moved to the end of the available memory, which decreases fragmentation. Some fragmentation is unavoidable over time, but the import optimisation definitely helps with avoiding as much initially.

Import Order Can Matter

If you have code that is failing on import, you may find that changing the import order allows your code to run. Before you have imported anything, there are larger chunks of memory available. Therefore, importing the larger libraries first, while there is still larger chunks of memory free, may enable your code to successfully run, even when it initially failed. This is not a guaranteed result, but in some cases, it can help.

Library Structure Not Intact or in Improper Directory

Remember that library files are available in two structures - a single file, and a directory containing multiple files. These structures must be kept intact for CircuitPython to be able to use the library. In the case of a single file, copy that standalone file to your CIRCUITPY drive. In the case of a directory, copy the entire directory and its contents to your CIRCUITPY drive.

Further, library files must not be placed into another directory. For example, if you create a neopixel/ directory on CIRCUITPY, and place neopixel.mpy inside of it, your code will fail. CircuitPython does not check superfluous directories for library files or packages.

In any of the situations above, your code will fail when you attempt to use the library in your code. When CircuitPython can't find the module you use within the library file, it will throw an error. The error will often be something like the following. (This error is specifically from an attempt to create the `pixels` object in code.py with neopixel.mpy in a neopixel/ directory.)

```
code.py output:
Traceback (most recent call last):
  File "code.py", line 4, in <module>
AttributeError: 'module' object has no attribute 'NeoPixel'

Code done running.
```

Don't Name Your Test Code the Same as a Library

Throughout your CircuitPython experience, you will write all sorts of different demos. You will obviously need to name files something other than code.py as you work through multiple demos, so you know what the file is, and so CircuitPython knows which file you currently want to run. However, in doing this, there is one serious caveat that can cause you all sorts of issues and make troubleshooting difficult. You must avoid naming your files the same name as a library file.

Say you write a NeoPixel example. Then you're ready to move on to a different example. So, you rename your current code.py to neopixel.py and move on to the next demo. The next time you need to use the NeoPixel library in your code, you will find that your code will fail. This is because it is trying to import your previously renamed code.py file, which of course does not have the library contents.

The best way to avoid this is to stay creative with your filenames. Do something like always include code in the name (i.e. neopixel_code.py), or get even more descriptive with it, calling it something like neopixel_all_pixels_red.py. These are two simple suggestions. There are also plenty of other ways to be creative about file naming.

PyCharm and CircuitPython

PyCharm () is a full-featured Python editor that includes super helpful things like code completion and error highlighting. It's available for free in a Community Edition.

These steps will help you make the most out of the PyCharm IDE for working with CircuitPython:

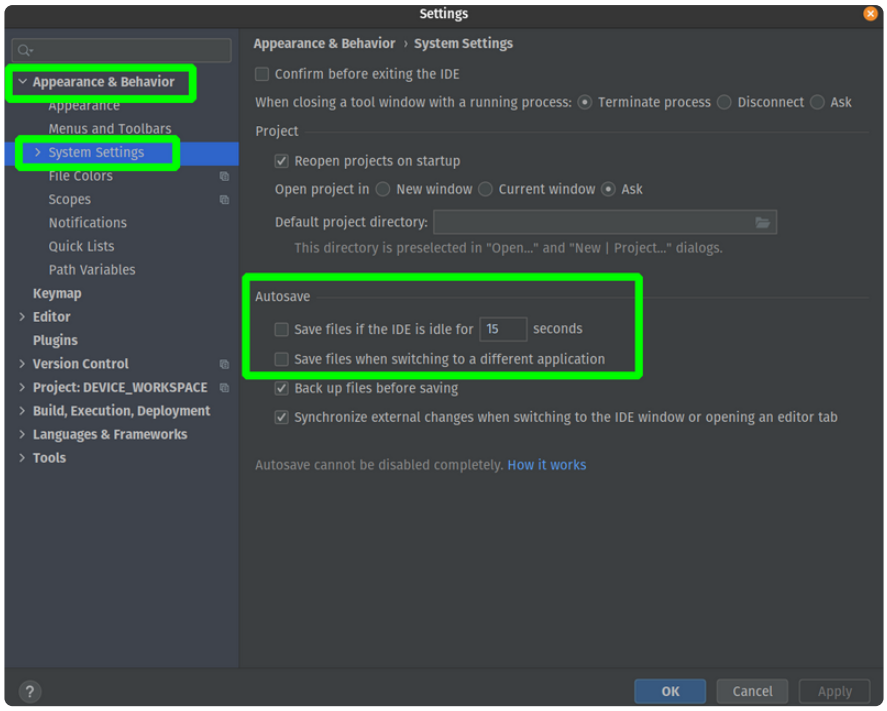
Disable PyCharm's auto-saving features ()
Create a project on your local system drive and add CIRCUITPY as a content root ()
Install the `circuitpython-stubs` ()
Install any libraries that you are working with if they are deployed to PyPi ()
Connect to serial console inside of the Terminal pane () (optional)

Disable Auto-save

By default, PyCharm will auto-save the code file(s) that you are working on very frequently. This is normally great, but with CIRCUITPY drives, it is can be undesirable because it will cause the device to reset and re-run the code more frequently than you might want. When you're using PyCharm to edit files directly on CIRCUITPY drives, it's best to disable the auto-save features and save manually with CTRL+S whenever you're ready to run the new version of code.py.

To disable auto-save, follow these steps:

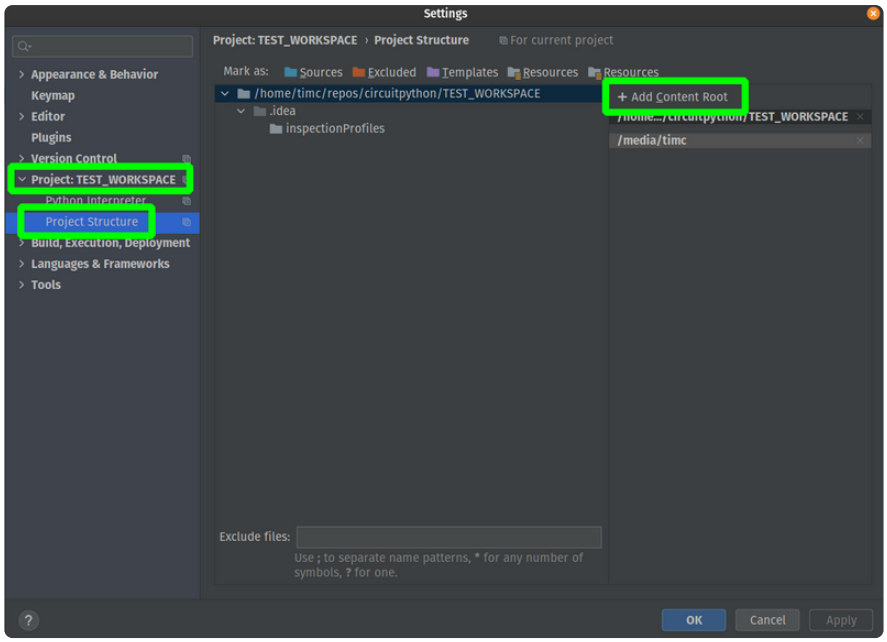
1. Open settings with CTRL+ALT+S or click File -> Settings
2. Open up Appearance & Behavior -> System Settings
3. Uncheck the box for "Save files if the IDE is idle for X seconds"
4. Uncheck the box for "Save files when switching to a different application"



Creating a project on a computer's file system

PyCharm likes to create a folder named .idea which holds configuration settings and meta information about your project. This folder gets created in the root directory of the project that you are working in. Ideally, we don't want this to end up on the CIRCUITPY drive, because it will eat up precious storage space and cause the device to reset when files inside of it are altered. The way we avoid this is by creating our project on the computer's hard drive and then adding CIRCUITPY or the directory that contains it as a "content root" so that PyCharm will show the files on the CIRCUITPY drive as part of the project.

1. Click File -> New Project
2. Enter a name for the project. I like to use DEVICE_WORKSPACE but you can choose any name you like
3. Enter or browse to a location on your computer's hard drive to store the project directory in
4. Click the Create button
5. With your new project open, click File -> Settings or press CTRL+ALT+S to open the settings window
6. Open "Project [YOUR_PROJECT_NAME]" then click "Project Structure"
7. Click the "+ Add Content Root" button on the right side of the window



The next step depends on your host OS.

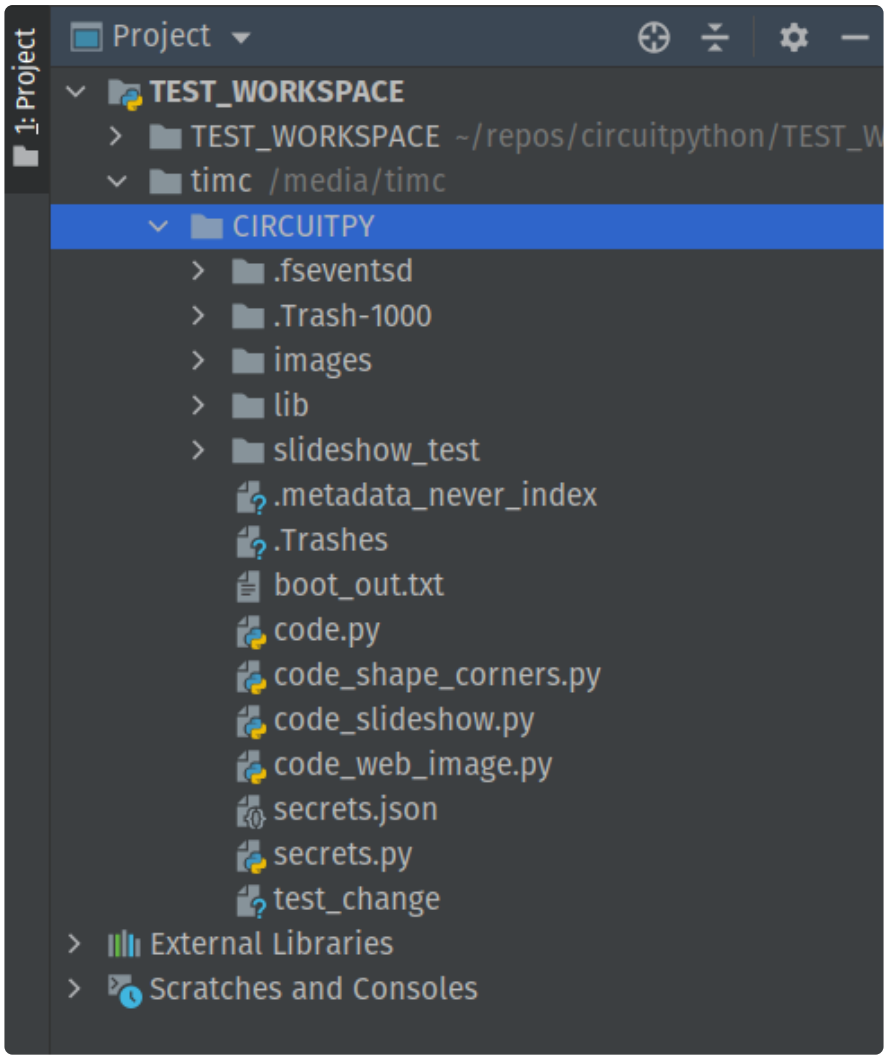
If you are on Linux, type or navigate to /media/[username]/ where [username] is your actual username on the computer. Click "Okay" to add this directory as the content root.

If you are on Mac, type or navigate to /Volumes/ then click "Okay" to add this directory as the content root.

If you are on Windows, type or navigate to the drive letter that matches your CIRCUIT PY drive, e.g. D:\, then click "Okay" to add this directory as the content root.

On Windows, the content root that you've added may eventually show up empty if you open the project when the CircuitPython device is not connected, or if the drive letter on your device changes due to other removable storage devices getting plugged in. Repeat the above instructions to re-add the device with its current drive letter. You can also click the "X" on the old / non-working content roots to remove them from the list in the same settings window.

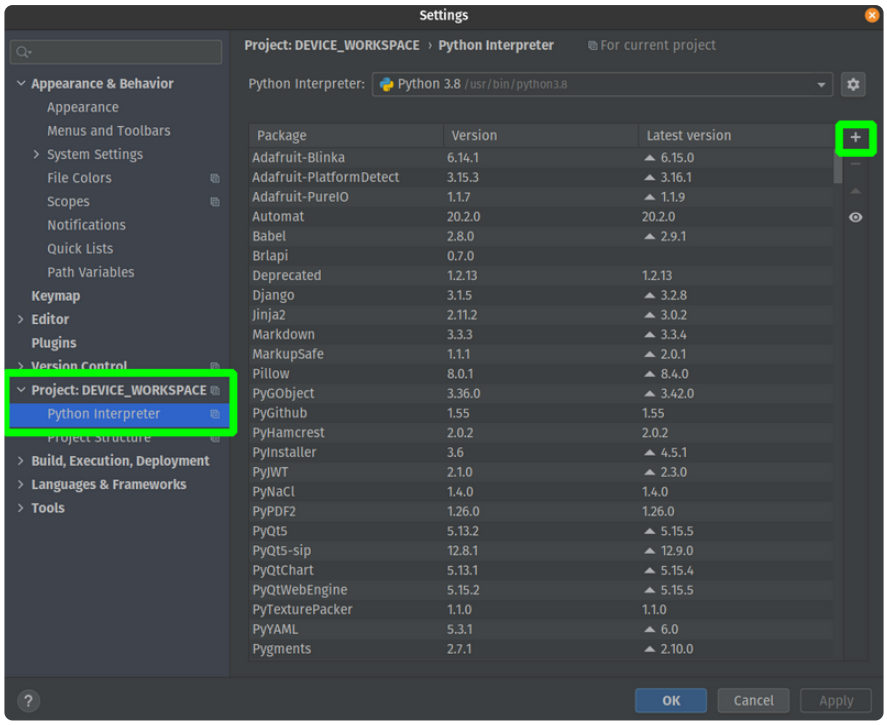
After completing these steps, you'll have a project on your computer's hard drive which will get the .idea directory stored inside of it. And you'll also have access to the CIRCUITPY drive(s) that are connected to your computer.

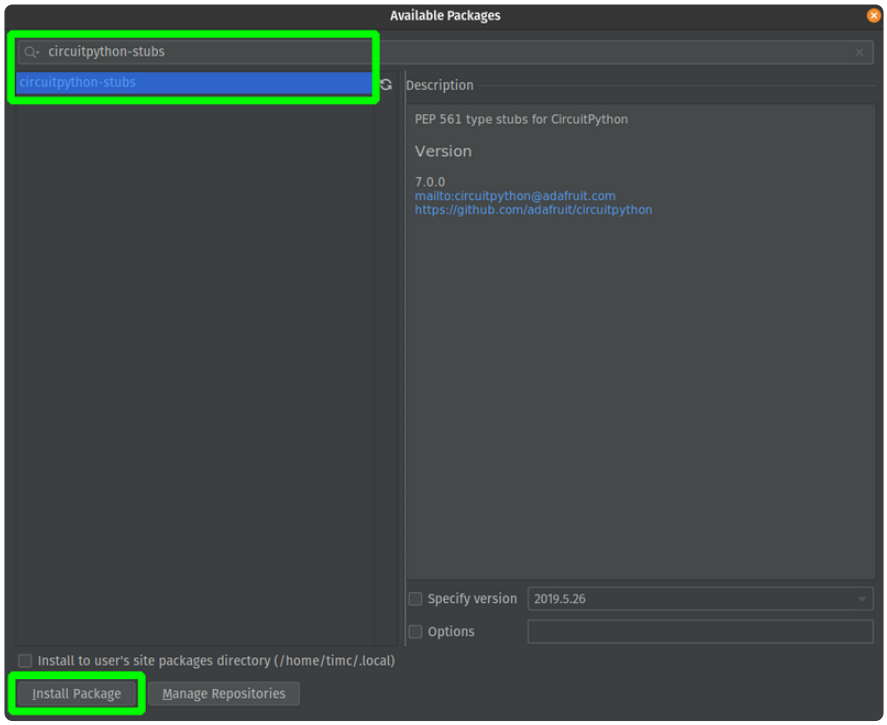


Install `circuitpython-stubs`

The `circuitpython-stubs` will let PyCharm know more information about the built-in core modules in CircuitPython so that it can offer relevant code hints and typing information. The stubs are published on PyPi, and installing them can be done in the PyCharm settings window.

1. Click File -> Settings or press CTRL+ALT+S to open the Settings window
2. Open "Project: [YOUR_PROJECT_NAME]" then click Python Interpreter
3. Click the + install button on the right side of the window
4. Type "circuitpython-stubs" into the search bar
5. Select the `circuitpython-stubs` package then press the Install Package button at the bottom left

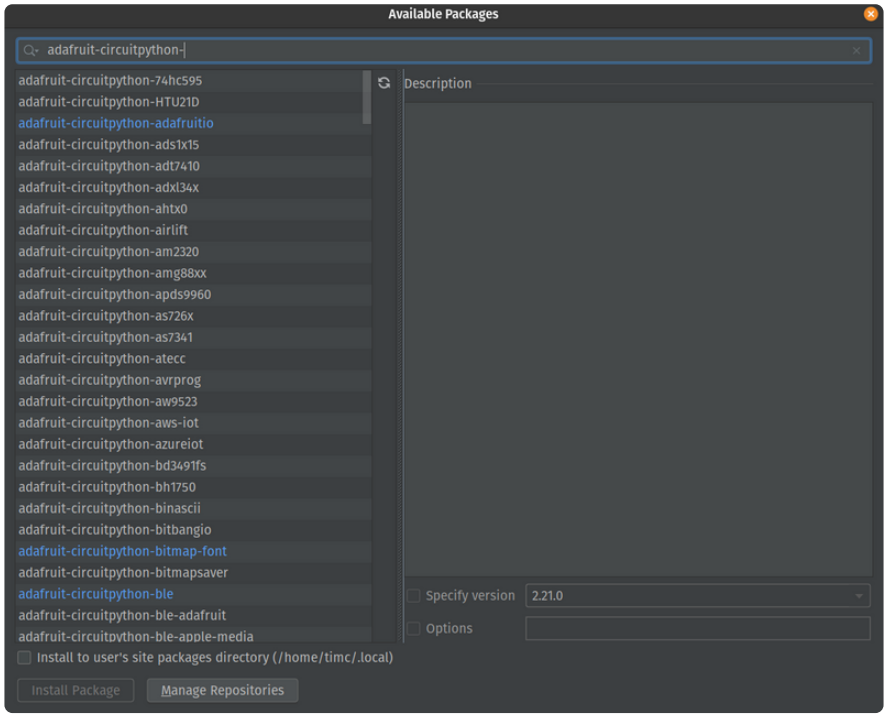




Install Libraries

If you project uses libraries from the CircuitPython Library Bundle, you can install them on your computer if they are published to PyPi. Doing so will give PyCharm information about the library in order to offer more relevant code hints and type information.

Use the above instructions to open the Available Packages window, then enter "adafruit-circuitpython-" into the search box to see all of the libraries deployed to PyPi.

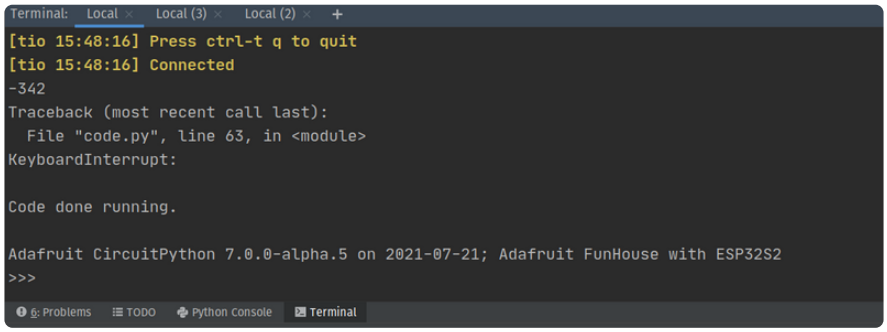


Serial console in the terminal pane

If you use the instructions from Advanced Serial Console on Mac () or Advanced Serial Console on Linux (), you can connect to the CircuitPython device's serial console inside the Terminal pane within PyCharm.

To open the Terminal pane, click View -> Tool Windows -> Terminal.

Connect to your device using `screen` or `tio`. Then, you can see output and interact with the device and REPL.



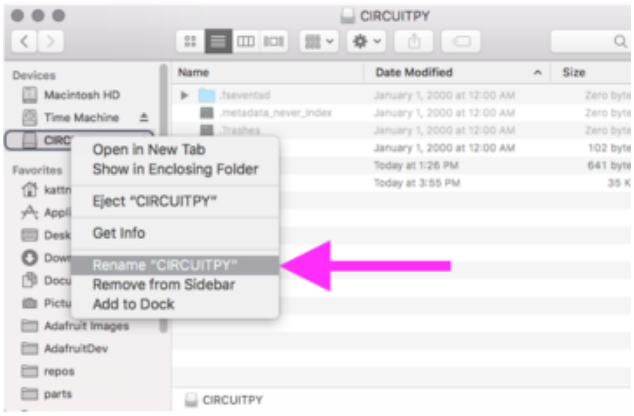
Renaming CIRCUITPY

There are many boards that work with CircuitPython. You may find yourself in a situation where you're working with more than one board at the same time. What happens when you have multiple boards plugged into your computer? You have multiple CIRCUITPY drives! How do you know which one is which? You can rename each CIRCUITPY drive to avoid confusion.

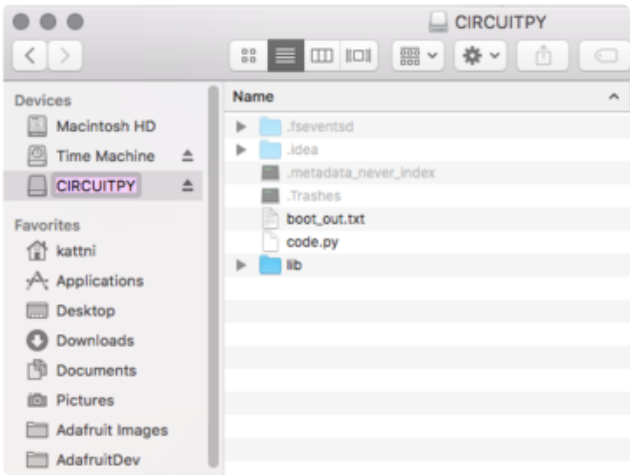
When you rename CIRCUITPY, it writes the name to the filesystem. This means that the name change will persist through disconnecting the board, as well as reloading CircuitPython!

The name must be 11 characters or less! This is a limitation of the filesystem. You will receive an error if you choose a name longer than 11 characters.

Renaming CIRCUITPY on Mac



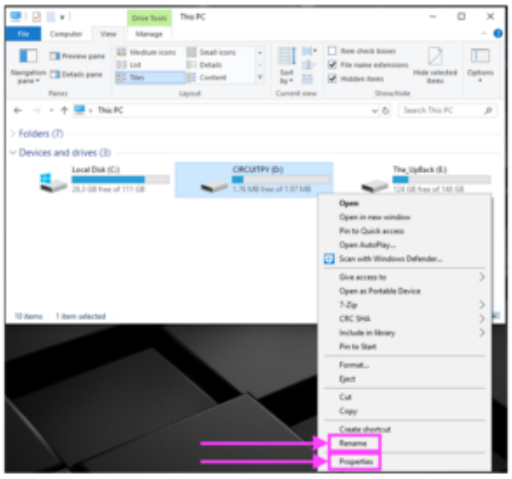
Renaming your CIRCUITPY drive on Mac is simple. Click on the drive in Finder so you can see the contents. Then, right click on the drive in Finder and choose "Rename".



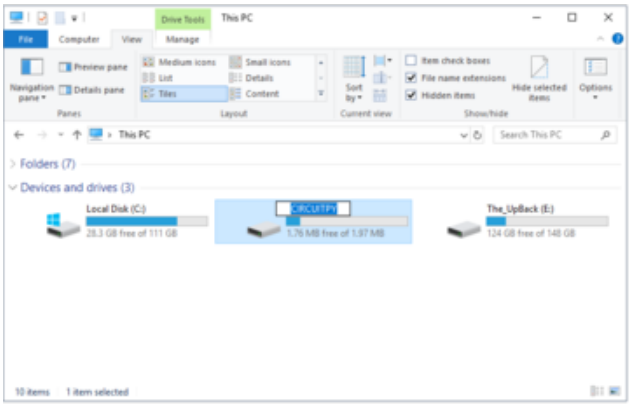
Once you click "Rename", in the right-click menu, the drive name will show up in a text box where you can rename the drive. Type in the new name.

Renaming CIRCUITPY on Windows

Renaming the CIRCUITPY drive on Windows is easy. Open File Explorer and find the C IRCUITPY drive. Right click on it, and click "Rename".



You can also rename the drive through the "Properties" menu, which can be opened several different ways through the Windows File Explorer, including right clicking on the CIRCUITPY drive and choosing "Properties".



Once you click "Rename" in the right-click menu, the drive name will show up in a text box where you can rename the drive. Type in the new name.

Renaming CIRCUITPY on Linux

Renaming CIRCUITPY on Linux requires a couple of steps. You'll need to identify the mount point, and then run a command to rename the drive.

Open a terminal program. Run the following to find out where your board is mounted:

```
df | grep CIRCUITPY
```

You will see CIRCUITPY on the right end of the resulting line. The dev/foo (where foo is the name of the mount point) on the left end of that line is the mount point.

Next you can run the following to unmount the board, replacing foo with your specific mount point.

```
sudo umount /dev/foo
```

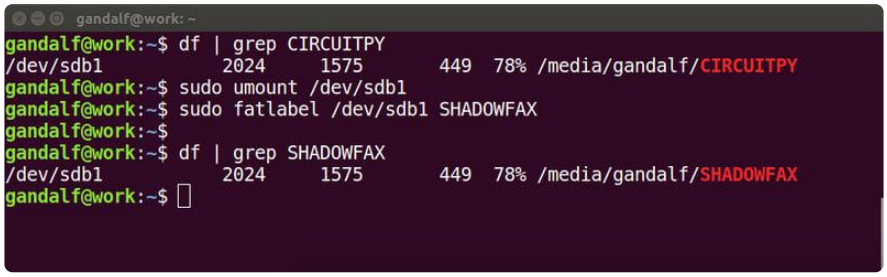
To rename the board, run the following:

```
sudo fatlabel /dev/foo NEW_NAME
```

Then, eject your board, unplug it, and plug it back in to force it to remount with the new name.

To check to see if it worked, look for the drive in your file manager. Or, you can run the following:

```
df | grep NEW_NAME
```



The name must be 11 characters or less! This is a limitation of the filesystem. You will receive an error if you choose a name longer than 11 characters.

Renaming CIRCUITPY through CircuitPython

You can also rename the board using CircuitPython. Create a new file on your CIRCUI TPY drive called boot.py. Copy the following code into the new boot.py file:

```
import storage

storage.remount("/", readonly=False)

m = storage.getmount("/")
m.label = "NEW_NAME"

storage.remount("/", readonly=True)

storage.enable_usb_drive()
```

Eject your board, and reboot the board either by pressing the reset button once, or unplugging it and plugging it back in. After a moment, it should show up in your file

explorer with the `NEW_NAME` you chose for it! You can delete boot.py after the newly named board shows up in your file explorer.

Reverting to CIRCUITPY

You can follow the same processes above to rename the drive back to CIRCUITPY.

You will also revert to CIRCUITPY by erasing the filesystem. If you are in a situation where you need to erase the filesystem on your CircuitPython board, the drive name will revert to CIRCUITPY on completion.

Advanced Serial Console on Windows

Windows 7 and 8.1

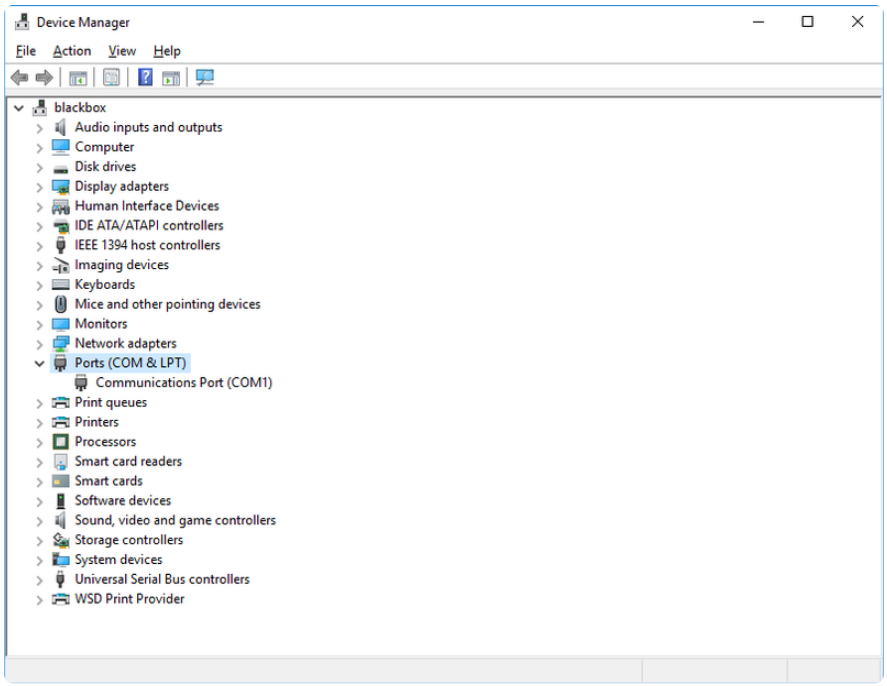
If you're using Windows 7 (or 8 or 8.1), you'll need to install drivers. See the Windows 7 and 8.1 Drivers page () for details. You will not need to install drivers on Mac, Linux or Windows 10.

You are strongly encouraged to upgrade to Windows 10 if you are still using Windows 7 or Windows 8 or 8.1. Windows 7 has reached end-of-life and no longer receives security updates. A free upgrade to Windows 10 is still available ().

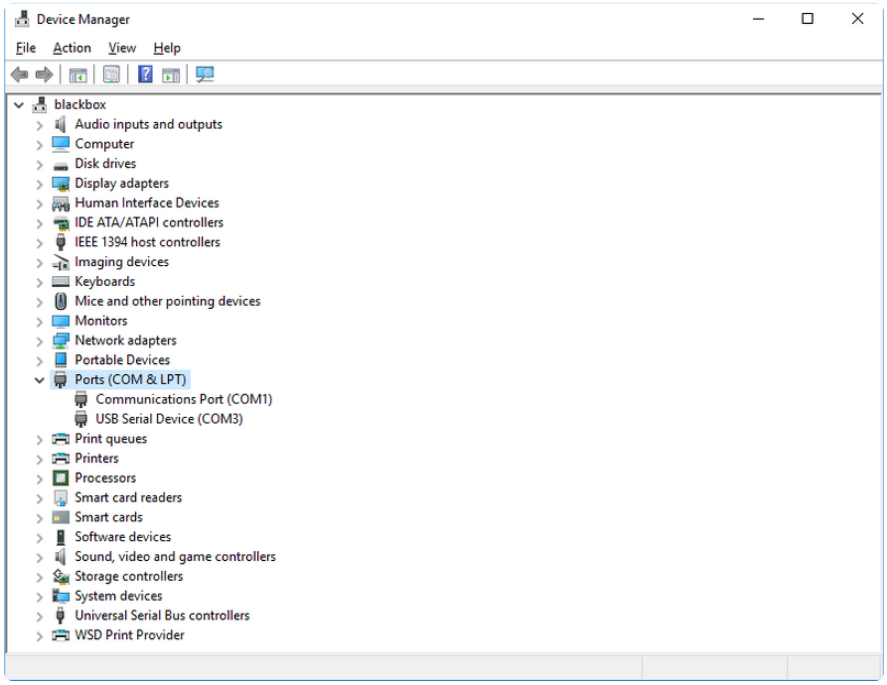
What's the COM?

First, you'll want to find out which serial port your board is using. When you plug your board in to USB on your computer, it connects to a serial port. The port is like a door through which your board can communicate with your computer using USB.

You'll use Windows Device Manager to determine which port the board is using. The easiest way to determine which port the board is using is to first check without the board plugged in. Open Device Manager. Click on Ports (COM & LPT). You should find something already in that list with (COM#) after it where # is a number.



Now plug in your board. The Device Manager list will refresh and a new item will appear under Ports (COM & LPT). You'll find a different (COM#) after this item in the list.



Sometimes the item will refer to the name of the board. Other times it may be called something like USB Serial Device, as seen in the image above. Either way, there is a new (COM#) following the name. This is the port your board is using.

Install Putty

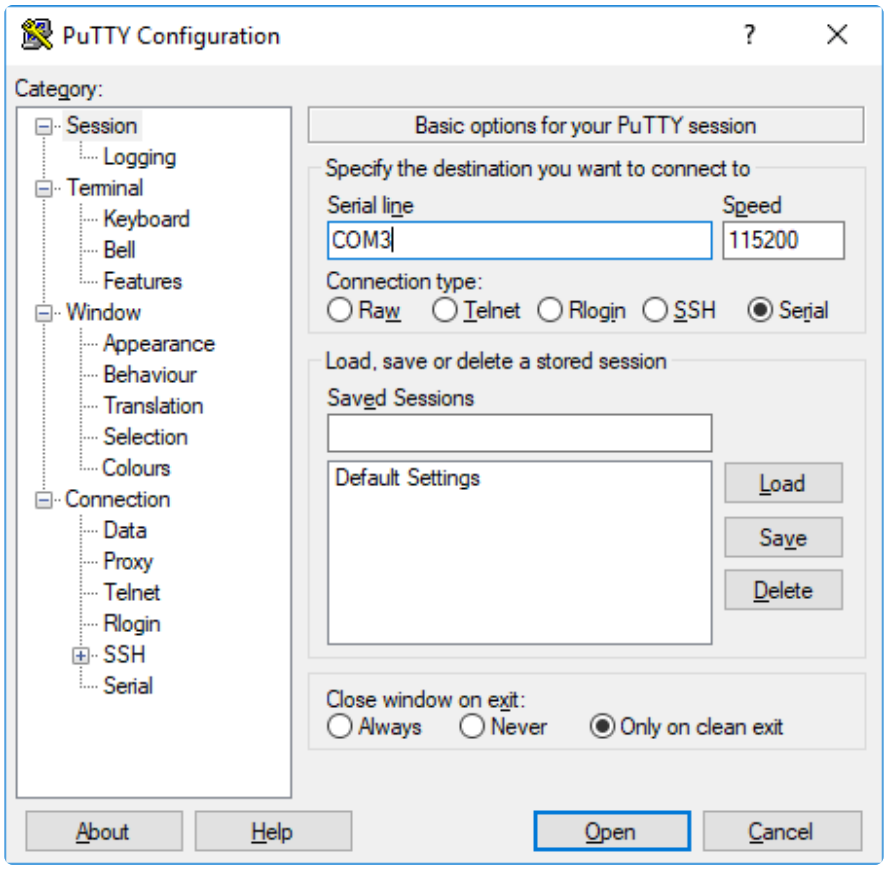
If you're using Windows, you'll need to download a terminal program. You're going to use PuTTY.

The first thing to do is download the latest version of PuTTY (). You'll want to download the Windows installer file. It is most likely that you'll need the 64-bit version. Download the file and install the program on your machine. If you run into issues, you can try downloading the 32-bit version instead. However, the 64-bit version will work on most PCs.

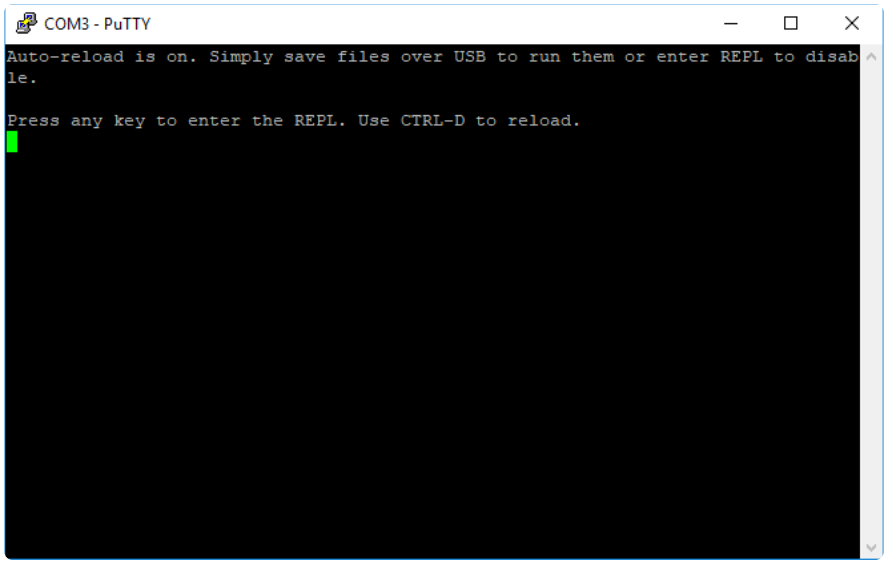
Now you need to open PuTTY.

- Under Connection type: choose the button next to Serial.
- In the box under Serial line, enter the serial port you found that your board is using.
- In the box under Speed, enter 115200. This called the baud rate, which is the speed in bits per second that data is sent over the serial connection. For boards with built in USB it doesn't matter so much but for ESP8266 and other board with a separate chip, the speed required by the board is 115200 bits per second. So you might as well just use 115200!

If you want to save those settings for later, use the options under Load, save or delete a stored session. Enter a name in the box under Saved Sessions, and click the Save button on the right.



Once your settings are entered, you're ready to connect to the serial console. Click "Open" at the bottom of the window. A new window will open.



If no code is running, the window will either be blank or will look like the window above. Now you're ready to see the results of your code.

Great job! You've connected to the serial console!

Advanced Serial Console on Mac

Connecting to the serial console on Mac does not require installing any drivers or extra software. You'll use a terminal program to find your board, and `screen` to connect to it. Terminal and `screen` both come installed by default.

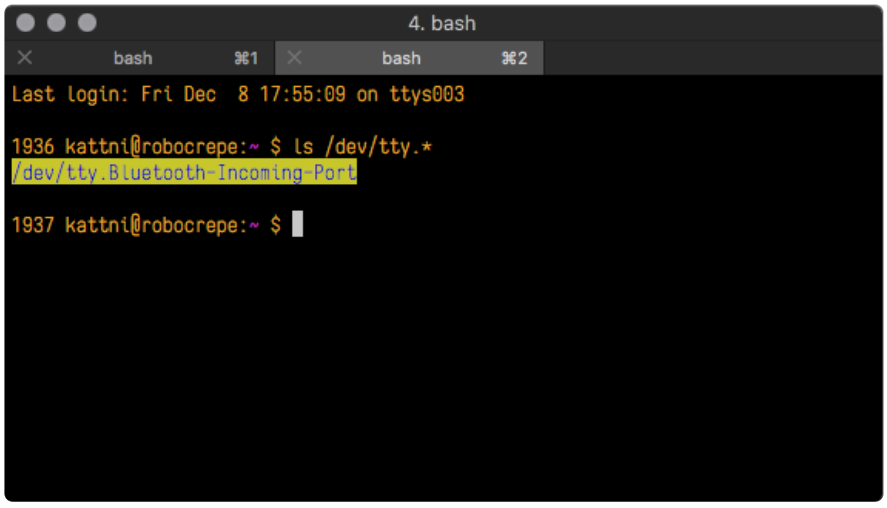
What's the Port?

First you'll want to find out which serial port your board is using. When you plug your board in to USB on your computer, it connects to a serial port. The port is like a door through which your board can communicate with your computer using USB.

The easiest way to determine which port the board is using is to first check without the board plugged in. Open Terminal and type the following:

```
ls /dev/tty.*
```

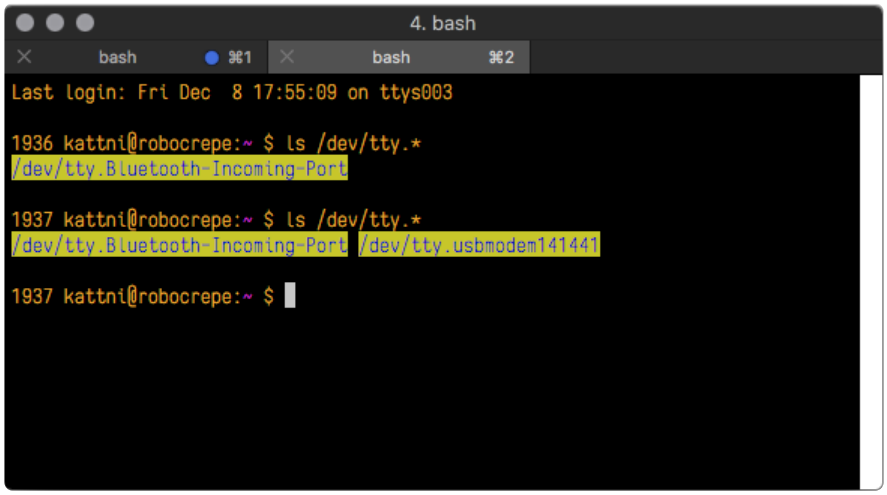
Each serial connection shows up in the `/dev/` directory. It has a name that starts with `tty.`. The command `ls` shows you a list of items in a directory. You can use `*` as a wildcard, to search for files that start with the same letters but end in something different. In this case, you're asking to see all of the listings in `/dev/` that start with `tty.` and end in anything. This will show us the current serial connections.



Now, plug your board. In Terminal, type:

```
ls /dev/tty.*
```

This will show you the current serial connections, which will now include your board.



A new listing has appeared called `/dev/tty.usbmodem141441`. The `tty.usbmodem141441` part of this listing is the name the example board is using. Yours will be called something similar.

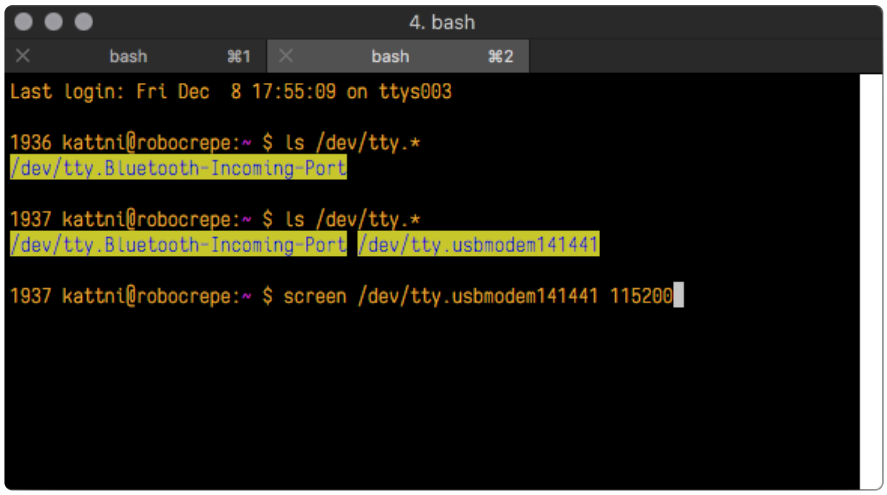
Using Linux, a new listing has appeared called `/dev/ttyACM0`. The `ttyACM0` part of this listing is the name the example board is using. Yours will be called something similar.

Connect with screen

Now that you know the name your board is using, you're ready connect to the serial console. You're going to use a command called `screen`. The `screen` command is included with MacOS. To connect to the serial console, use Terminal. Type the following command, replacing `board_name` with the name you found your board is using:

```
screen /dev/tty.board_name 115200
```

The first part of this establishes using the `screen` command. The second part tells screen the name of the board you're trying to use. The third part tells screen what baud rate to use for the serial connection. The baud rate is the speed in bits per second that data is sent over the serial connection. In this case, the speed required by the board is 115200 bits per second.



Press enter to run the command. It will open in the same window. If no code is running, the window will be blank. Otherwise, you'll see the output of your code.

Great job! You've connected to the serial console!

Advanced Serial Console on Linux

Connecting to the serial console on Linux does not require installing any drivers, but you may need to install `screen` using your package manager. You'll use a terminal program to find your board, and `screen` to connect to it. There are a variety of terminal programs such as gnome-terminal (called Terminal) or Konsole on KDE.

The `tio` program works as well to connect to your board, and has the benefit of automatically reconnecting. You would need to install it using your package manager.

What's the Port?

First you'll want to find out which serial port your board is using. When you plug your board in to USB on your computer, it connects to a serial port. The port is like a door through which your board can communicate with your computer using USB.

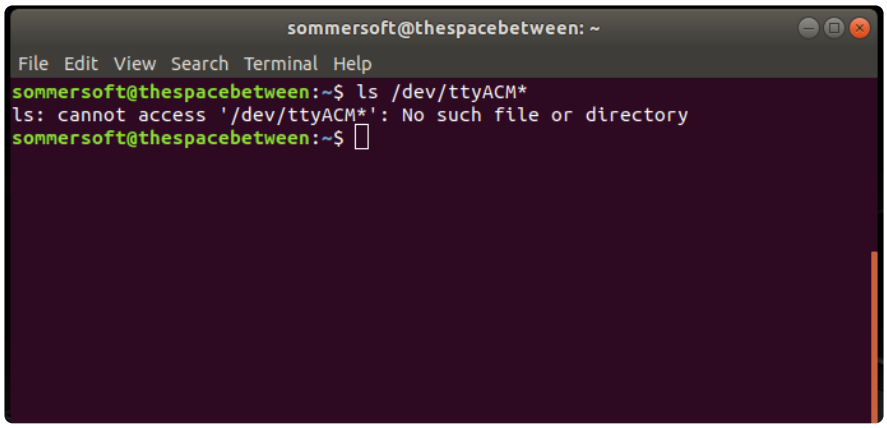
The easiest way to determine which port the board is using is to first check without the board plugged in. Open your terminal program and type the following:

```
ls /dev/ttyACM*
```

Each serial connection shows up in the /dev/ directory. It has a name that starts with tt yACM. The command `ls` shows you a list of items in a directory. You can use `*` as a wildcard, to search for files that start with the same letters but end in something

different. In this case, You're asking to see all of the listings in /dev/ that start with ttyA CM and end in anything. This will show us the current serial connections.

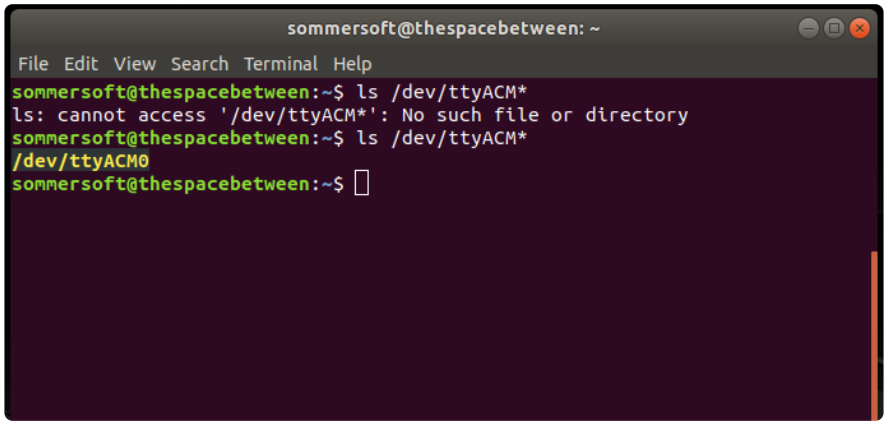
In the example below, the error is indicating that are no current serial connections starting with ttyACM.



Now plug in your board. In your terminal program, type:

```
ls /dev/ttyACM*
```

This will show you the current serial connections, which will now include your board.



A new listing has appeared called /dev/ttyACM0. The ttyACM0 part of this listing is the name the example board is using. Yours will be called something similar.

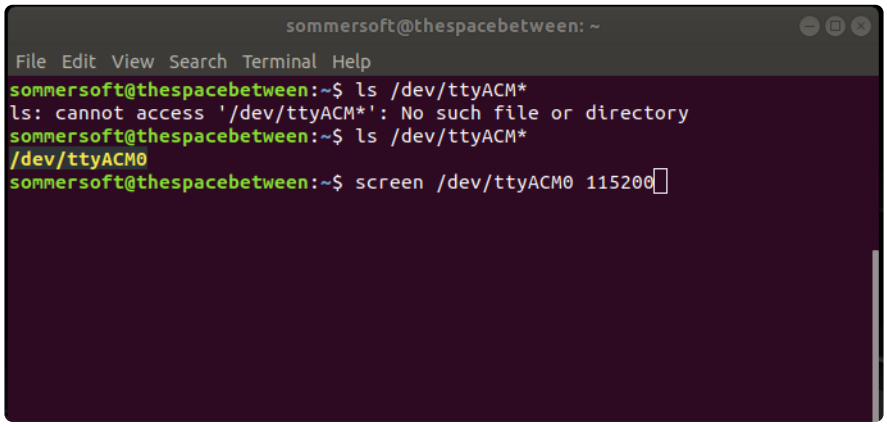
Connect with screen

Now that you know the name your board is using, you're ready connect to the serial console. You'll use a command called `screen`. You may need to install it using the package manager.

To connect to the serial console, use your terminal program. Type the following command, replacing `board_name` with the name you found your board is using:

```
screen /dev/tty.board_name 115200
```

The first part of this establishes using the `screen` command. The second part tells screen the name of the board you're trying to use. The third part tells screen what baud rate to use for the serial connection. The baud rate is the speed in bits per second that data is sent over the serial connection. In this case, the speed required by the board is 115200 bits per second.

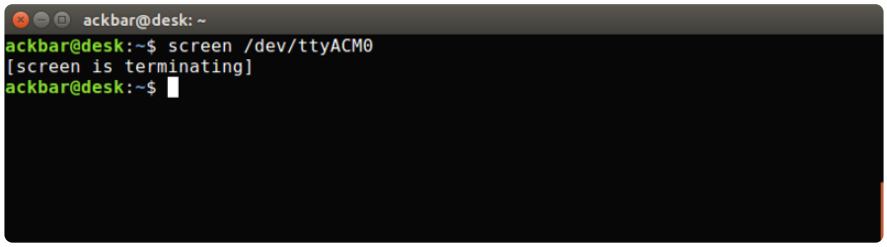


Press enter to run the command. It will open in the same window. If no code is running, the window will be blank. Otherwise, you'll see the output of your code.

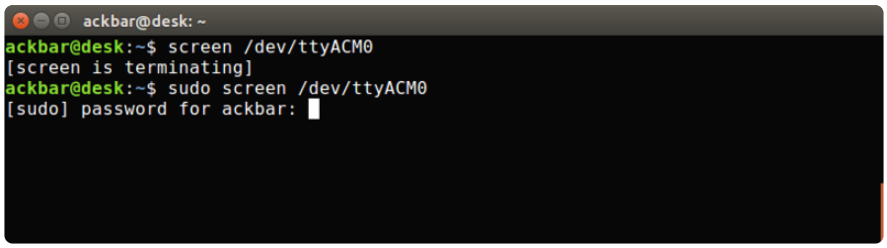
Great job! You've connected to the serial console!

Permissions on Linux

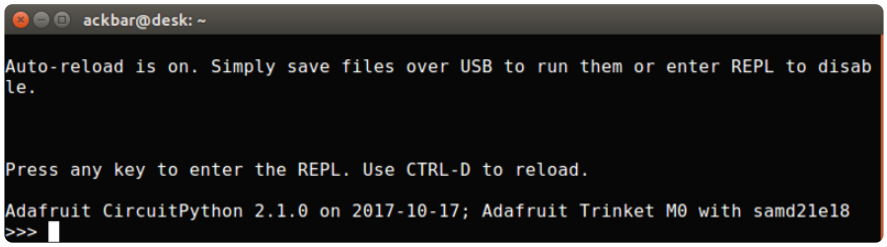
If you try to run `screen` and it doesn't work, then you may be running into an issue with permissions. Linux keeps track of users and groups and what they are allowed to do and not do, like access the hardware associated with the serial connection for running `screen`. So if you see something like this:



then you may need to grant yourself access. There are generally two ways you can do this. The first is to just run `screen` using the `sudo` command, which temporarily gives you elevated privileges.

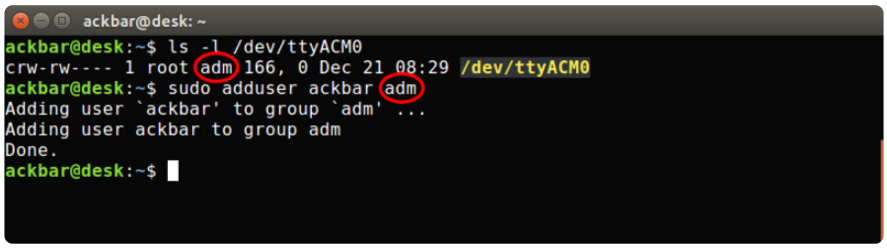


Once you enter your password, you should be in:

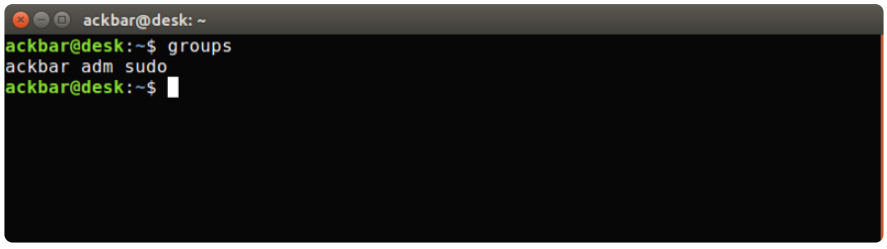


The second way is to add yourself to the group associated with the hardware. To figure out what that group is, use the command `ls -l` as shown below. The group name is circled in red.

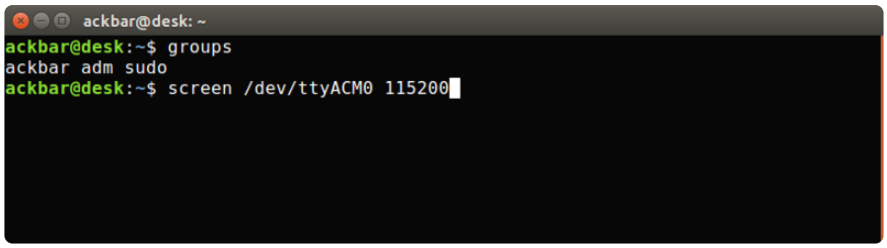
Then use the command `adduser` to add yourself to that group. You need elevated privileges to do this, so you'll need to use `sudo`. In the example below, the group is a dm and the user is ackbar.



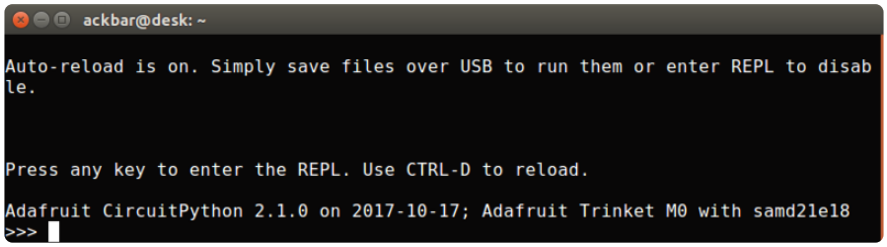
After you add yourself to the group, you'll need to logout and log back in, or in some cases, reboot your machine. After you log in again, verify that you have been added to the group using the command `groups`. If you are still not in the group, reboot and check again.



And now you should be able to run `screen` without using `sudo`.



And you're in:



The examples above use `screen`, but you can also use other programs, such as `putty` or `picocom`, if you prefer.

Non-UF2 Installation

This installation page is only required if you do not have UF2 bootloader installed (e.g. boardBOOT drag-n-drop)!

Espressif Boards without UF2 Bootloaders

See this page () for how to load a .bin file on an ESP32 or ESP32-C3 board.

STM Boards

See this page () for how to upload a .bin file using the built-in STM bootloader.

Flashing with Bossac - For Non-Express Feather M0's & Arduino Zero

The older Feather M0 boards don't come with UF2, instead they come with a simpler bootloader called bossa. This is also what is installed on Arduino Zero's and other CircuitPython compatible boards that use the ATSAMDx1 or nRF52840. It is the only method you can use if your CircuitPython installation file is a .bin rather than a .uf2

Command-Line ahoy!

Flashing with bossac requires the use of your computer's command line interface. On Windows, that's the cmd or powershell tool. On Mac and Linux, use Terminal!

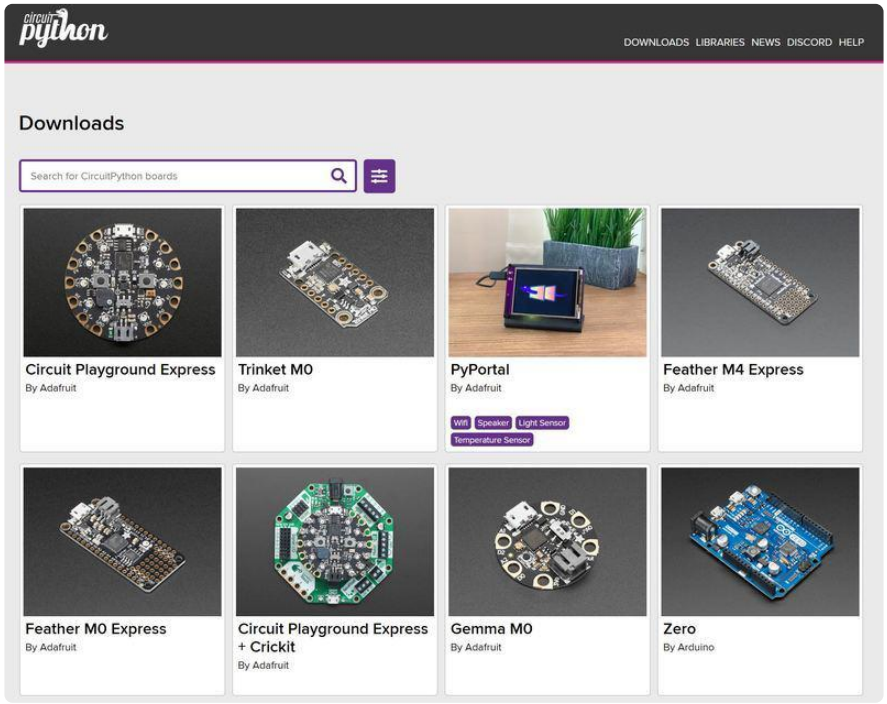
Download Latest CircuitPython Firmware

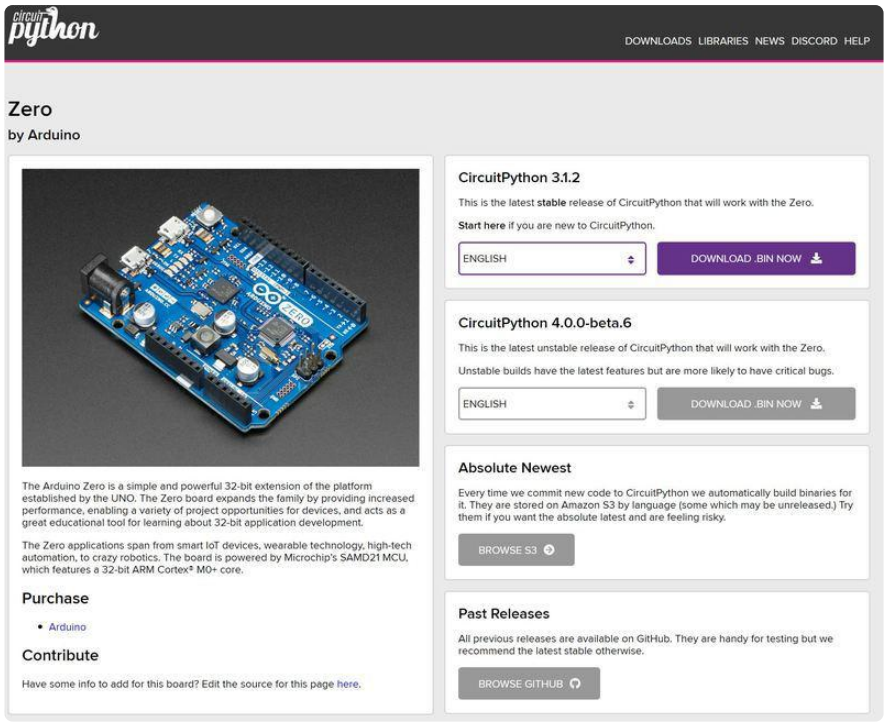
The first thing you'll want to do is download the most recent version of CircuitPython.

Compatible boards have .bin files available for download on circuitpython.org ().

Click the green button below to search for your board to find the appropriate file.

Find and download software for your board from CircuitPython.org





Once downloaded, save the .bin file onto your desktop, you'll need it soon!

If you are running Windows 7, you must install the driver set (it is covered on this page) () so you have access to the COM port.

Download BOSSA

Once you have a firmware image you'll need to download the BOSSA tool, which that can load firmware on SAMD21/51 boards. This tool is actually used internally by the Arduino IDE when it programs these boards, however you can use it yourself to load custom firmware

Be aware you must use 1.7.0 or higher version of bossac to program SAMD21 and SAMD51 boards! Versions of BOSSA before version 1.7.0 won't work because it doesn't support the SAMD21/51 chips. Also note that bossac versions 1.9.0 and newer have incompatibly changed their command-line parameters, and can erase your bootloader if it is not protected. (Adafruit boards ship with protected bootloaders.) Follow the directions below carefully, depending on which version you have.

To flash with `bossac` (BOSSA's command line tool) first download the latest version from here (). The `ming32` version is for Windows, `apple-darwin` for Mac OSX and various `linux` options for Linux.

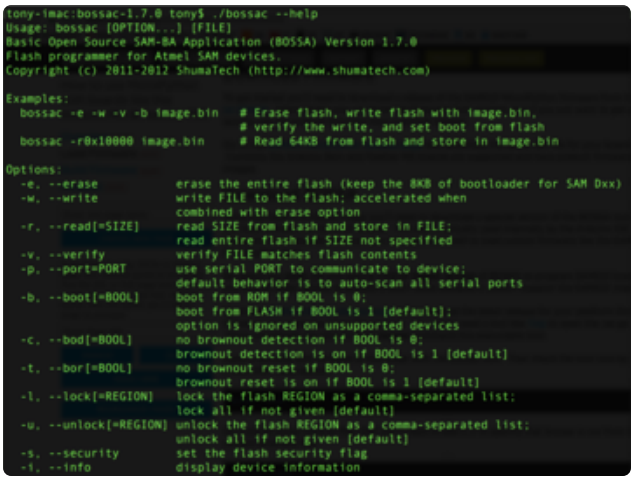
`bossac` works with .bin files only, it will not work with .uf2 files!

Never use bossac or its Windows equivalent (BOSSA) 1.9.0 or higher without specifying an offset: 0x2000 for SAMD21, 0x4000 for SAMD51. Omitting the offset will damage the fuse settings on the board and can lead to a broken bootloader. See the link below.

erase option wipes SAMD21 protected bootloader, fuses

Test bossac

Open a terminal and navigate to the folder with the `bossac` tool. Then check the tool runs by running it with the `--help` option:



Open a terminal and navigate to the folder with the `bossac` tool. Then check the tool runs by running it with the --help option with `bossac --help`

Or if you're using Linux or Mac OSX you'll need to add a ./ to specify that bossac is run from the current directory like `./bossac --help`

Make sure you see BOSSA version 1.7.0 or higher! And see the warning below about version 1.9.0 and higher. If you see a lower version then you accidentally downloaded an older version of the tool and it won't work to flash SAMD21 chips. Go back and grab the latest release from this BOSSA GitHub repository () as mentioned above.

Port Selection for Mac OS

You'll need to know what port to use if you're on a Mac.

In your same terminal window, run the command `ls /dev/cu.*`. Note the ports listed, then plug in your board and run the command again. The device may be listed something like /dev/cu.usbmodem14301. Make note of the port name for the `bossac` section below.

Get Into the Bootloader

You'll have to 'kick' the board into the bootloader manually. Do so by double-clicking the reset button. The red "#13" LED should pulse on and off. If you are using an Arduino Zero, make sure you are connected to the native USB port not the debugging/programming port.

One special note, if you're using the Arduino M0 from Arduino.org you'll need to replace its bootloader with the Arduino Zero bootloader so it can work with BOSSA. To do this install the Arduino/Genuino Zero board in the Arduino IDE board manager and then follow these steps to burn the Arduino Zero bootloader () (using the program ming port on the board). Once you've loaded the Arduino Zero bootloader you should be able to use the M0 with bossac as described below.

Run the bossac Command

With your board plugged in and running the bootloader you're ready to flash CircuitPython firmware onto the board. Copy the firmware .bin file to the same directory as the `bossac` tool, then in a terminal navigate to that location and run one ofthe following commands, depending on which version of `bossac` you have.

With bossac versions 1.9 or later, you must use the --offset parameter on the command line, and it must have the correct value for your board. Otherwise you will brick your board.

With bossac version 1.9 or later, you must give an **`--offset`** parameter on the command line to specify where to start writing the firmware in flash memory. This parameter was added in bossac 1.8.0 with a default of `0x2000`, but starting in 1.9, the default offset was changed to `0x0000`, which is not what you want in most cases. If you omit the argument for bossac 1.9 or later, you will probably see a "Verify Failed" error from bossac, and you will damage the fuse settings on your board, rendering the bootloader inoperable. Remember to change the option for `-p` or `--port` to match the port on your machine.

Replace the filename below with the name of your downloaded **`.bin`**: it will vary based on your board!

Using bossac Version 1.7.0

There is no `--offset` parameter available. Use a command line like this:

MacOS

```
bossac -p /dev/cu.usbmodemxxx -e -w -v -R adafruit-circuitpython-boardname-version.bin
```

Linux

The port to use will vary, but will probably be `/dev/ttyACM0`.

```
bossac -p /dev/ttyACMx -e -w -v -R adafruit-circuitpython-boardname-version.bin
```

Windows

The COM port you use will vary. Choose the one the for the board (see the Device Manager if necessary).

```
bossac -p COMx -e -w -v -R adafruit-circuitpython-boardname-version.bin
```

For example,

```
bossac -p /dev/cu.usbmodem14301 -e -w -v -R adafruit-circuitpython-feather_m0_express-3.0.0.bin
```

Using bossac Versions 1.8, and 1.9 or Later

For M0 (SAMD21) boards, which have an 8kB bootloader, you must specify `--offset=0x2000`. Do not omit the **`--offset`** parameter; you will brick your board if you omit it.

MacOS

```
bossac -p /dev/cu.usbmodemxxx -e -w -v -R --offset=0x2000 adafruit-circuitpython-boardname-version.bin
```

Linux

The port to use will vary, but will probably be `/dev/ttyACM0`.

```
bossac -p /dev/ttyACMx -e -w -v -R --offset=0x2000 adafruit-circuitpython-boardname-version.bin
```

Windows

The COM port you use will vary. Choose the one the for the board (see the Device Manager if necessary).

```
bossac -p COMx -e -w -v -R --offset=0x2000 adafruit-circuitpython-boardname-version.bin
```

For example, on MacOS:

```
bossac -p /dev/cu.usbmodem14301 -e -w -v -R --offset=0x2000 adafruit-circuitpython-feather_m0_express-3.0.0.bin
```

For M4 (SAMD51) boards, which have a 16kB bootloader, you must specify `-offset=0x4000`. Do not omit the **`--offset`** parameter; you will brick your board if you omit it.

MacOS

```
bossac -p /dev/cu.usbmodemxxx -e -w -v -R --offset=0x4000 adafruit-circuitpython-boardname-version.bin
```

Linux

The port to use will vary, but will probably be `/dev/ttyACM0`.

```
bossac -p /dev/ttyACMx -e -w -v -R --offset=0x4000 adafruit-circuitpython-boardname-version.bin
```

Windows

The COM port you use will vary. Choose the one the for the board (see the Device Manager if necessary).

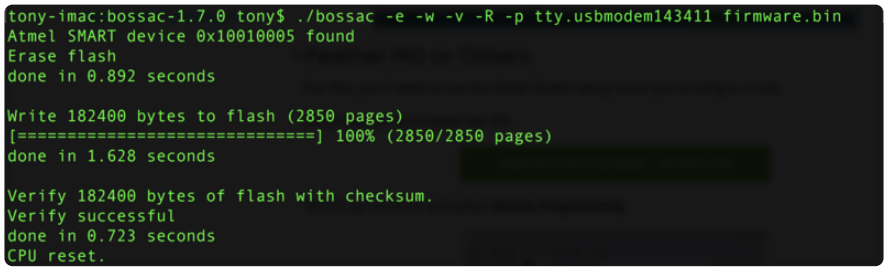
```
bossac -p COM5 -e -w -v -R --offset=0x4000 adafruit-circuitpython-boardname-version.bin
```

For example, on MacOS:

`bossac -p /dev/cu.usbmodem14301 -e -w -v -R --offset=0x4000 adafruit-circuitpython-feather_m4_express-3.0.0.bin`

This will `e` rase the chip, `w` rite the given file, `v` erify the write and `R` eset the board. On Linux or MacOS you may need to run this command with `sudo ./bossac ...`, or add yourself to the dialout group first.

After bossac loads the firmware you should see output similar to the following:



After reset, CircuitPython should be running and the CIRCUITPY drive will appear. You can always manually reset the board by clicking the reset button, sometimes that is needed to get it to 'wake up'. Express boards may cause a warning of an early eject of a USB drive but just ignore it. Nothing important was being written to the drive!

Frequently Asked Questions

These are some of the common questions regarding CircuitPython and CircuitPython microcontrollers.

What are some common acronyms to know?

CP or CPy = CircuitPython ()
CPC = Circuit Playground Classic () (does not run CircuitPython)
CPX = Circuit Playground Express ()
CPB = Circuit Playground Bluefruit ()

Using Older Versions

As CircuitPython development continues and there are new releases, Adafruit will stop supporting older releases. Visit https://circuitpython.org/downloads to download the latest version of CircuitPython for your board. You must download the CircuitPython Library Bundle that matches your version of CircuitPython. Please update CircuitPython and then visit https://circuitpython.org/libraries to download the latest Library Bundle.

I have to continue using CircuitPython 7.x or earlier. Where can I find compatible libraries?

We are no longer building or supporting the CircuitPython 7.x or earlier library bundles. We highly encourage you to update CircuitPython to the latest version () and use the current version of the libraries (). However, if for some reason you cannot update, here are the last available library bundles for older versions:

- 2.x bundle ()
- 3.x bundle ()
- 4.x bundle ()
- 5.x bundle ()
- 6.x bundle ()
- 7.x bundle ()

Python Arithmetic

Does CircuitPython support floating-point numbers?

All CircuitPython boards support floating point arithmetic, even if the microcontroller chip does not support floating point in hardware. Floating point numbers are stored in 30 bits, with an 8-bit exponent and a 22-bit mantissa. Note that this is two bits less than standard 32-bit single-precision floats. You will get about 5-1/2 digits of decimal precision.

(The broadcom port may provide 64-bit floats in some cases.)

Does CircuitPython support long integers, like regular Python?

Python long integers (integers of arbitrary size) are available on most builds, except those on boards with the smallest available firmware size. On these boards, integers are stored in 31 bits.

Boards without long integer support are mostly SAMD21 ("M0") boards without an external flash chip, such as the Adafruit Gemma M0, Trinket M0, QT Py M0, and the Trinkey series. There are also a number of third-party boards in this category. There are also a few small STM third-party boards without long integer support.

`time.localtime()`, `time.mktime()`, `time.time()`, and `time.monotonic_ns()` are available only on builds with long integers.

Wireless Connectivity

How do I connect to the Internet with CircuitPython?

If you'd like to include WiFi in your project, your best bet is to use a board that is running natively on ESP32 chipsets - those have WiFi built in!

If your development board has an SPI port and at least 4 additional pins, you can check out this guide () on using AirLift with CircuitPython - extra wiring is required and some boards like the MacroPad or NeoTrellis do not have enough available pins to add the hardware support.

For further project examples, and guides about using AirLift with specific hardware, check out the Adafruit Learn System ().

How do I do BLE (Bluetooth Low Energy) with CircuitPython?

The nRF52840 and nRF52833 boards have the most complete BLE implementation. Your program can act as both a BLE central and peripheral. As a central, you can scan for advertisements, and connect to an advertising board. As a peripheral, you can advertise, and you can create services available to a central. Pairing and bonding are supported.

ESP32-C3 and ESP32-S3 boards currently provide an incomplete () BLE implementation. Your program can act as a central, and connect to a peripheral. You can advertise, but you cannot create services. You cannot advertise anonymously. Pairing and bonding are not supported.

The ESP32 could provide a similar implementation, but it is not yet available. Note that the ESP32-S2 does not have Bluetooth capability.

On most other boards with adequate firmware space, BLE is available for use with AirLift () or other NINA-FW-based co-processors. Some boards have this coprocessor on board, such as the PyPortal (). Currently, this implementation only supports acting as a BLE peripheral. Scanning and connecting as a central are not yet implemented. Bonding and pairing are not supported.

Are there other ways to communicate by radio with CircuitPython?

Check out Adafruit's RFM boards ()for simple radio communication supported by CircuitPython, which can be used over distances of 100m to over a km, depending on the version. The RFM SAMD21 M0 boards can be used, but they were not designed for CircuitPython, and have limited RAM and flash space; using the RFM breakouts or FeatherWings with more capable boards will be easier.

Asyncio and Interrupts

Is there asyncio support in CircuitPython?

There is support for asyncio starting with CircuitPython 7.1.0, on all boards except the smallest SAMD21 builds. Read about using it in the Cooperative Multitasking in CircuitPython () Guide.

Does CircuitPython support interrupts?

No. CircuitPython does not currently support interrupts - please use asyncio for multitasking / 'threaded' control of your code

---

Status RGB LED

My RGB NeoPixel/DotStar LED is blinking funny colors - what does it mean?

The status LED can tell you what's going on with your CircuitPython board. Read more here for what the colors mean! ()

Memory Issues

What is a MemoryError?

Memory allocation errors happen when you're trying to store too much on the board. The CircuitPython microcontroller boards have a limited amount of memory available. You can have about 250 lines of code on the M0 Express boards. If you try to `import` too many libraries, a combination of large libraries, or run a program with too many lines of code, your code will fail to run and you will receive a `MemoryError` in the serial console.

What do I do when I encounter a MemoryError?

Try resetting your board. Each time you reset the board, it reallocates the memory. While this is unlikely to resolve your issue, it's a simple step and is worth trying.

Make sure you are using .mpy versions of libraries. All of the CircuitPython libraries are available in the bundle in a .mpy format which takes up less memory than .py format. Be sure that you're using the latest library bundle () for your version of CircuitPython.

If that does not resolve your issue, try shortening your code. Shorten comments, remove extraneous or unneeded code, or any other clean up you can do to shorten your code. If you're using a lot of functions, you could try moving those into a separate library, creating a .mpy of that library, and importing it into your code.

You can turn your entire file into a .mpy and `import` that into code.py. This means you will be unable to edit your code live on the board, but it can save you space.

Can the order of my `import` statements affect memory?

It can because the memory gets fragmented differently depending on allocation order and the size of objects. Loading .mpy files uses less memory so its recommended to do that for files you aren't editing.

How can I create my own .mpy files?

You can make your own .mpy versions of files with `mpy-cross`.

You can download `mpy-cross` for your operating system from here (). Builds are available for Windows, macOS, x64 Linux, and Raspberry Pi Linux. Choose the latest `mpy-cross` whose version matches the version of CircuitPython you are using.

To make a .mpy file, run `./mpy-cross path/to/yourfile.py` to create a yourfile.mpy in the same directory as the original file.

How do I check how much memory I have free?

Run the following to see the number of bytes available for use:

```
import gc
gc.mem_free()
```

Unsupported Hardware

Is ESP8266 or ESP32 supported in CircuitPython? Why not?

We dropped ESP8266 support as of 4.x - For more information please read about it here ()!

As of CircuitPython 8.x we have started to support ESP32 and ESP32-C3 and have added a WiFi workflow for wireless coding! ()

We also support ESP32-S2 & ESP32-S3, which have native USB.

Does Feather M0 support WINC1500?

No, WINC1500 will not fit into the M0 flash space.

Can AVRs such as ATmega328 or ATmega2560 run CircuitPython?

No.

Troubleshooting

From time to time, you will run into issues when working with CircuitPython. Here are a few things you may encounter and how to resolve them.

As CircuitPython development continues and there are new releases, Adafruit will stop supporting older releases. Visit https://circuitpython.org/downloads to download the latest version of CircuitPython for your board. You must download the CircuitPython Library Bundle that matches your version of CircuitPython. Please update CircuitPython and then visit https://circuitpython.org/libraries to download the latest Library Bundle.

Always Run the Latest Version of CircuitPython and Libraries

As CircuitPython development continues and there are new releases, Adafruit will stop supporting older releases. You need to update to the latest CircuitPython. ().

You need to download the CircuitPython Library Bundle that matches your version of CircuitPython. Please update CircuitPython and then download the latest bundle ().

As new versions of CircuitPython are released, Adafruit will stop providing the previous bundles as automatically created downloads on the Adafruit CircuitPython Library Bundle repo. If you must continue to use an earlier version, you can still download the appropriate version of `mpy-cross` from the particular release of CircuitPython on the CircuitPython repo and create your own compatible .mpy library files. However, it is best to update to the latest for both CircuitPython and the library bundle.

I have to continue using CircuitPython 7.x or earlier. Where can I find compatible libraries?

Adafruit is no longer building or supporting the CircuitPython 7.x or earlier library bundles. You are highly encourged to update CircuitPython to the latest version () and use the current version of the libraries (). However, if for some reason you cannot update, links to the previous bundles are available in the FAQ ().

Bootloader (boardnameBOOT) Drive Not Present

You may have a different board.

Only Adafruit Express boards and the SAMD21 non-Express boards ship with the UF2 bootloader ()installed. The Feather M0 Basic, Feather M0 Adalogger, and similar boards use a regular Arduino-compatible bootloader, which does not show a boardna meBOOT drive.

MakeCode

If you are running a MakeCode () program on Circuit Playground Express, press the reset button just once to get the CPLAYBOOT drive to show up. Pressing it twice will not work.

macOS

DriveDx and its accompanything SAT SMART Driver can interfere with seeing the BOOT drive. See this forum post () for how to fix the problem.

Windows 10

Did you install the Adafruit Windows Drivers package by mistake, or did you upgrade to Windows 10 with the driver package installed? You don't need to install this package on Windows 10 for most Adafruit boards. The old version (v1.5) can interfere with recognizing your device. Go to Settings -> Apps and uninstall all the "Adafruit" driver programs.

Windows 7 or 8.1

To use a CircuitPython-compatible board with Windows 7 or 8.1, you must install a driver. Installation instructions are available here ().

It is recommended () that you upgrade to Windows 10 if possible; an upgrade is probably still free for you. Check here ().

The Windows Drivers installer was last updated in November 2020 (v2.5.0.0) . Windows 7 drivers for CircuitPython boards released since then, including RP2040 boards, are not available. There are no plans to release drivers for new boards. The boards work fine on Windows 10.

You should now be done! Test by unplugging and replugging the board. You should see the CIRCUITPY drive, and when you double-click the reset button (single click on Circuit Playground Express running MakeCode), you should see the appropriate boar dnameBOOT drive.

Let us know in the Adafruit support forums () or on the Adafruit Discord () if this does not work for you!

Windows Explorer Locks Up When Accessing boardnameBOOT Drive

On Windows, several third-party programs that can cause issues. The symptom is that you try to access the boardnameBOOT drive, and Windows or Windows Explorer seems to lock up. These programs are known to cause trouble:

- AIDA64: to fix, stop the program. This problem has been reported to AIDA64. They acquired hardware to test, and released a beta version that fixes the problem. This may have been incorporated into the latest release. Please let us know in the forums if you test this.
- Hard Disk Sentinel
- Kaspersky anti-virus: To fix, you may need to disable Kaspersky completely. Disabling some aspects of Kaspersky does not always solve the problem. This problem has been reported to Kaspersky.
- ESET NOD32 anti-virus: There have been problems with at least version 9.0.386.0, solved by uninstallation.

Copying UF2 to boardnameBOOT Drive Hangs at 0% Copied

On Windows, a Western DIgital (WD) utility that comes with their external USB drives can interfere with copying UF2 files to the boardnameBOOT drive. Uninstall that utility to fix the problem.

CIRCUITPY Drive Does Not Appear or Disappears Quickly

Kaspersky anti-virus can block the appearance of the CIRCUITPY drive. There has not yet been settings change discovered that prevents this. Complete uninstallation of Kaspersky fixes the problem.

Norton anti-virus can interfere with CIRCUITPY. A user has reported this problem on Windows 7. The user turned off both Smart Firewall and Auto Protect, and CIRCUITPY then appeared.

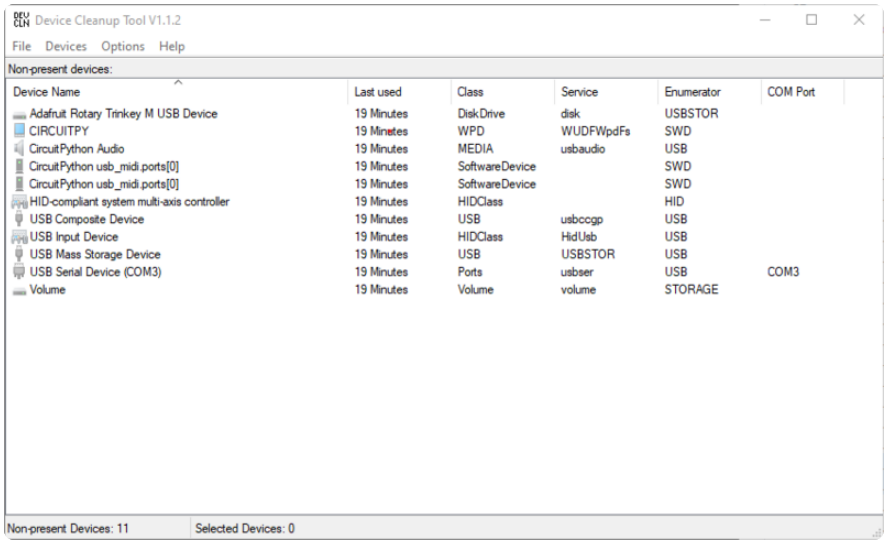
Sophos Endpoint security software can cause CIRCUITPY to disappear () and the BOOT drive to reappear. It is not clear what causes this behavior.

Samsung Magician can cause CIRCUITPY to disappear (reported here () and here ()).

Device Errors or Problems on Windows

Windows can become confused about USB device installations. This is particularly true of Windows 7 and 8.1. It is recommended () that you upgrade to Windows 10 if possible; an upgrade is probably still free for you: see this link ().

If not, try cleaning up your USB devices. Use Uwe Sieber's Device Cleanup Tool () (on that page, scroll down to "Device Cleanup Tool"). Download and unzip the tool. Unplug all the boards and other USB devices you want to clean up. Run the tool as Administrator. You will see a listing like this, probably with many more devices. It is listing all the USB devices that are not currently attached.



Select all the devices you want to remove, and then press Delete. It is usually safe just to select everything. Any device that is removed will get a fresh install when you plug it in. Using the Device Cleanup Tool also discards all the COM port assignments for the unplugged boards. If you have used many Arduino and CircuitPython boards, you have probably seen higher and higher COM port numbers used, seemingly without end. This will fix that problem.

Serial Console in Mu Not Displaying Anything

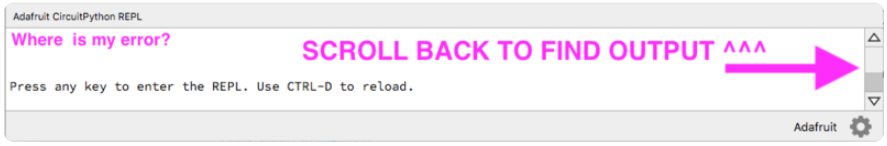
There are times when the serial console will accurately not display anything, such as, when no code is currently running, or when code with no serial output is already running before you open the console. However, if you find yourself in a situation where you feel it should be displaying something like an error, consider the following.

Depending on the size of your screen or Mu window, when you open the serial console, the serial console panel may be very small. This can be a problem. A basic CircuitPython error takes 10 lines to display!

```
Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
Traceback (most recent call last):
  File "code.py", line 7
SyntaxError: invalid syntax


Press any key to enter the REPL. Use CTRL-D to reload.
```

More complex errors take even more lines!

Therefore, if your serial console panel is five lines tall or less, you may only see blank lines or blank lines followed by `Press any key to enter the REPL. Use CTRL-D to reload.`. If this is the case, you need to either mouse over the top of the panel to utilise the option to resize the serial panel, or use the scrollbar on the right side to scroll up and find your message.



This applies to any kind of serial output whether it be error messages or print statements. So before you start trying to debug your problem on the hardware side, be sure to check that you haven't simply missed the serial messages due to serial output panel height.

code.py Restarts Constantly

CircuitPython will restart code.py if you or your computer writes to something on the CIRCUITPY drive. This feature is called auto-reload, and lets you test a change to your program immediately.

Some utility programs, such as backup, anti-virus, or disk-checking apps, will write to the CIRCUITPY as part of their operation. Sometimes they do this very frequently, causing constant restarts.

Acronis True Image and related Acronis programs on Windows are known to cause this problem. It is possible to prevent this by disabling the " ()Acronis Managed Machine Service Mini" ().

If you cannot stop whatever is causing the writes, you can disable auto-reload by putting this code in boot.py or code.py:

```
import supervisor

supervisor.runtime.autoreload = False
```

CircuitPython RGB Status Light

Nearly all CircuitPython-capable boards have a single NeoPixel or DotStar RGB LED on the board that indicates the status of CircuitPython. A few boards designed before CircuitPython existed, such as the Feather M0 Basic, do not.

Circuit Playground Express and Circuit Playground Bluefruit have multiple RGB LEDs, but do NOT have a status LED. The LEDs are all green when in the bootloader. In versions before 7.0.0, they do NOT indicate any status while running CircuitPython.

CircuitPython 7.0.0 and Later

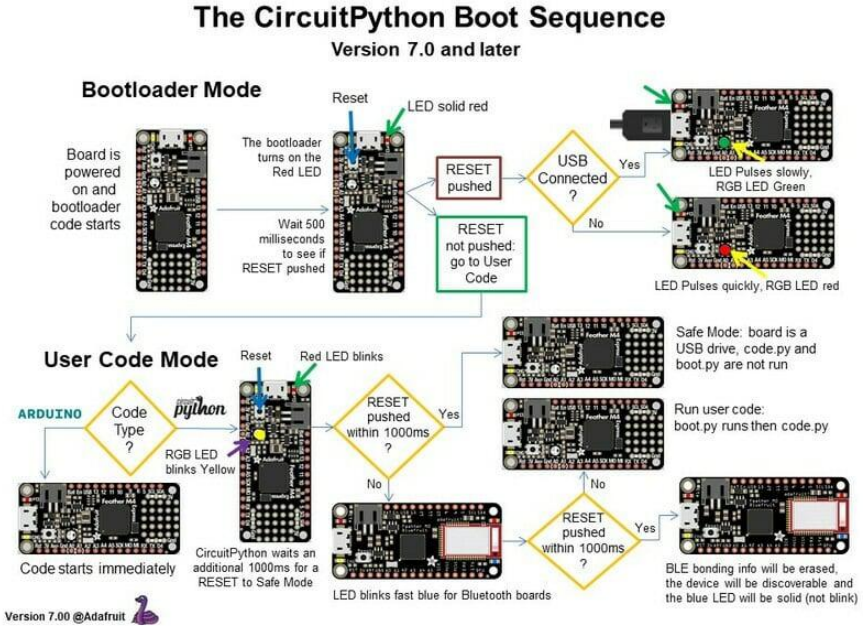
The status LED blinks were changed in CircuitPython 7.0.0 in order to save battery power and simplify the blinks. These blink patterns will occur on single color LEDs when the board does not have any RGB LEDs. Speed and blink count also vary for this reason.

On start up, the LED will blink YELLOW multiple times for 1 second. Pressing the RESET button (or on Espressif, the BOOT button) during this time will restart the board and then enter safe mode. On Bluetooth capable boards, after the yellow blinks, there will be a set of faster blue blinks. Pressing reset during the BLUE blinks will clear Bluetooth information and start the device in discoverable mode, so it can be used with a BLE code editor.

Once started, CircuitPython will blink a pattern every 5 seconds when no user code is running to indicate why the code stopped:

- 1 GREEN blink: Code finished without error.
- 2 RED blinks: Code ended due to an exception. Check the serial console for details.
- 3 YELLOW blinks: CircuitPython is in safe mode. No user code was run. Check the serial console for safe mode reason.

When in the REPL, CircuitPython will set the status LED to WHITE. You can change the LED color from the REPL. The status indicator will not persist on non-NeoPixel or DotStar LEDs.



CircuitPython 6.3.0 and earlier

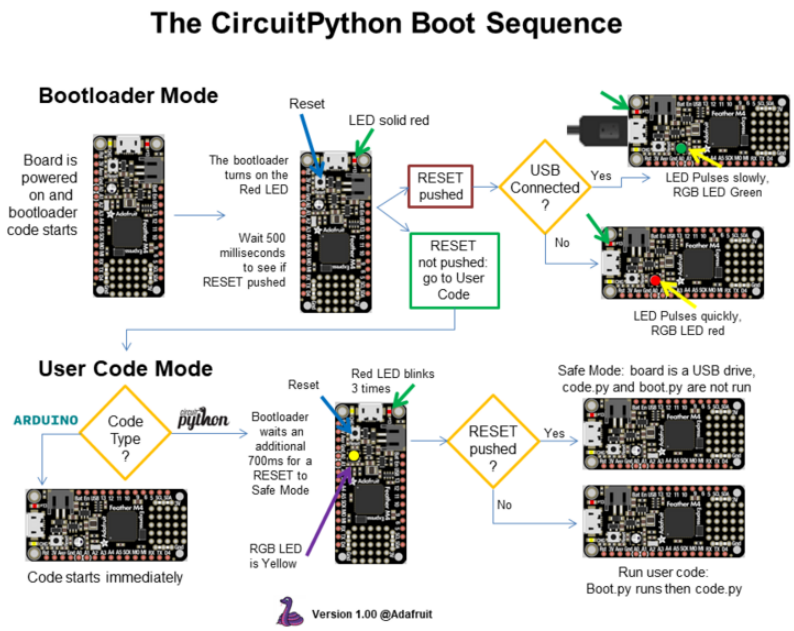
Here's what the colors and blinking mean:

- steady GREEN: code.py (or code.txt, main.py, or main.txt) is running
- pulsing GREEN: code.py (etc.) has finished or does not exist
- steady YELLOW at start up: (4.0.0-alpha.5 and newer) CircuitPython is waiting for a reset to indicate that it should start in safe mode
- pulsing YELLOW: Circuit Python is in safe mode: it crashed and restarted
- steady WHITE: REPL is running
- steady BLUE: boot.py is running

Colors with multiple flashes following indicate a Python exception and then indicate the line number of the error. The color of the first flash indicates the type of error:

- GREEN: IndentationError
- CYAN: SyntaxError
- WHITE: NameError
- ORANGE: OSError
- PURPLE: ValueError
- YELLOW: other error

These are followed by flashes indicating the line number, including place value. WHITE flashes are thousands' place, BLUE are hundreds' place, YELLOW are tens' place, and CYAN are one's place. So for example, an error on line 32 would flash YELLOW three times and then CYAN two times. Zeroes are indicated by an extra-long dark gap.



Serial console showing `ValueError: Incompatible .mpy file`

This error occurs when importing a module that is stored as a .mpy binary file that was generated by a different version of CircuitPython than the one its being loaded into. In particular, the mpy binary format changed between CircuitPython versions 6.x and 7.x, 2.x and 3.x, and 1.x and 2.x.

So, for instance, if you upgraded to CircuitPython 7.x from 6.x you'll need to download a newer version of the library that triggered the error on `import`. All libraries are available in the Adafruit bundle ().

CIRCUITPY Drive Issues

You may find that you can no longer save files to your CIRCUITPY drive. You may find that your CIRCUITPY stops showing up in your file explorer, or shows up as NO_NAME. These are indicators that your filesystem has issues. When the CIRCUITPY disk is not safely ejected before being reset by the button or being disconnected from USB, it may corrupt the flash drive. It can happen on Windows, Mac or Linux, though it is more common on Windows.

Be aware, if you have used Arduino to program your board, CircuitPython is no longer able to provide the USB services. You will need to reload CircuitPython to resolve this situation.

The easiest first step is to reload CircuitPython. Double-tap reset on the board so you get a boardnameBOOT drive rather than a CIRCUITPY drive, and copy the latest version of CircuitPython (.uf2) back to the board. This may restore CIRCUITPY functionality.

If reloading CircuitPython does not resolve your issue, the next step is to try putting the board into safe mode.

Safe Mode

Whether you've run into a situation where you can no longer edit your code.py on your CIRCUITPY drive, your board has gotten into a state where CIRCUITPY is read-only, or you have turned off the CIRCUITPY drive altogether, safe mode can help.

Safe mode in CircuitPython does not run any user code on startup, and disables auto-reload. This means a few things. First, safe mode bypasses any code in boot.py (where you can set CIRCUITPY read-only or turn it off completely). Second, it does not run the code in code.py. And finally, it does not automatically soft-reload when data is written to the CIRCUITPY drive.

Therefore, whatever you may have done to put your board in a non-interactive state, safe mode gives you the opportunity to correct it without losing all of the data on the CIRCUITPY drive.

Entering Safe Mode in CircuitPython 7.x and Later

To enter safe mode when using CircuitPython 7.x, plug in your board or hit reset (highlighted in red above). Immediately after the board starts up or resets, it waits 1000ms. On some boards, the onboard status LED will blink yellow during that time. If you press reset during that 1000ms, the board will start up in safe mode. It can be difficult to react to the yellow LED, so you may want to think of it simply as a "slow" double click of the reset button. (Remember, a fast double click of reset enters the bootloader.)

Entering Safe Mode in CircuitPython 6.x

To enter safe mode when using CircuitPython 6.x, plug in your board or hit reset (highlighted in red above). Immediately after the board starts up or resets, it waits 700ms. On some boards, the onboard status LED (highlighted in green above) will turn solid yellow during this time. If you press reset during that 700ms, the board will start up in safe mode. It can be difficult to react to the yellow LED, so you may want to think of it simply as a slow double click of the reset button. (Remember, a fast double click of reset enters the bootloader.)

In Safe Mode

Once you've entered safe mode successfully in CircuitPython 6.x, the LED will pulse yellow.

If you successfully enter safe mode on CircuitPython 7.x, the LED will intermittently blink yellow three times.

If you connect to the serial console, you'll find the following message.

```
Auto-reload is off.
Running in safe mode! Not running saved code.

CircuitPython is in safe mode because you pressed the reset button during boot.
Press again to exit safe mode.

Press any key to enter the REPL. Use CTRL-D to reload.
```

You can now edit the contents of the CIRCUITPY drive. Remember, your code will not run until you press the reset button, or unplug and plug in your board, to get out of safe mode.

At this point, you'll want to remove any user code in code.py and, if present, the boot.py file from CIRCUITPY. Once removed, tap the reset button, or unplug and plug in your board, to restart CircuitPython. This will restart the board and may resolve your drive issues. If resolved, you can begin coding again as usual.

If safe mode does not resolve your issue, the board must be completely erased and CircuitPython must be reloaded onto the board.

You WILL lose everything on the board when you complete the following steps. If possible, make a copy of your code before continuing.

To erase CIRCUITPY: `storage.erase_filesystem()`

CircuitPython includes a built-in function to erase and reformat the filesystem. If you have a version of CircuitPython older than 2.3.0 on your board, you can update to the newest version () to do this.

1. Connect to the CircuitPython REPL () using Mu or a terminal program.
2. Type the following into the REPL:

```
&gt;&gt;&gt; import storage
&gt;&gt;&gt; storage.erase_filesystem()
```

CIRCUITPY will be erased and reformatted, and your board will restart. That's it!

Erase CIRCUITPY Without Access to the REPL

If you can't access the REPL, or you're running a version of CircuitPython previous to 2.3.0 and you don't want to upgrade, there are options available for some specific boards.

The options listed below are considered to be the "old way" of erasing your board. The method shown above using the REPL is highly recommended as the best method for erasing your board.

If at all possible, it is recommended to use the REPL to erase your CIRCUITPY drive. The REPL method is explained above.

For the specific boards listed below:

If the board you are trying to erase is listed below, follow the steps to use the file to erase your board.

1. Download the correct erase file:

Circuit Playground Express

Feather M0 Express

Feather M4 Express

Metro M0 Express

Metro M4 Express QSPI Eraser

Trellis M4 Express (QSPI)

Grand Central M4 Express (QSPI)

PyPortal M4 Express (QSPI)

Circuit Playground Bluefruit (QSPI)

Monster M4SK (QSPI)

PyBadge/PyGamer QSPI Eraser.UF2

CLUE_Flash_Erase.UF2

Matrix_Portal_M4_(QSPI).UF2

RP2040 boards (flash_nuke.uf2)

2. Double-click the reset button on the board to bring up the boardnameBOOT drive.
3. Drag the erase .uf2 file to the boardnameBOOT drive.
4. The status LED will turn yellow or blue, indicating the erase has started.
5. After approximately 15 seconds, the status LED will light up green. On the NeoTrellis M4 this is the first NeoPixel on the grid
6. Double-click the reset button on the board to bring up the boardnameBOOT d rive.
7. Drag the appropriate latest release of CircuitPython () .uf2 file to the boardnam eBOOT drive.

It should reboot automatically and you should see CIRCUITPY in your file explorer again.

If the LED flashes red during step 5, it means the erase has failed. Repeat the steps starting with 2.

If you haven't already downloaded the latest release of CircuitPython for your board, check out the installation page (). You'll also need to load your code and reinstall your libraries!

For SAMD21 non-Express boards that have a UF2 bootloader:

Any SAMD21-based microcontroller that does not have external flash available is considered a SAMD21 non-Express board. Non-Express boards that have a UF2 bootloader include Trinket M0, GEMMA M0, QT Py M0, and the SAMD21-based Trinkey boards.

If you are trying to erase a SAMD21 non-Express board, follow these steps to erase your board.

1. Download the erase file:

SAMD21 non-Express Boards

2. Double-click the reset button on the board to bring up the boardnameBOOT drive.
3. Drag the erase .uf2 file to the boardnameBOOT drive.
4. The boot LED will start flashing again, and the boardnameBOOT drive will reappear.
5. Drag the appropriate latest release CircuitPython () .uf2 file to the boardnameBOOT drive.

It should reboot automatically and you should see CIRCUITPY in your file explorer again.

If you haven't already downloaded the latest release of CircuitPython for your board, check out the installation page () YYou'll also need to load your code and reinstall your libraries!

For SAMD21 non-Express boards that do not have a UF2 bootloader:

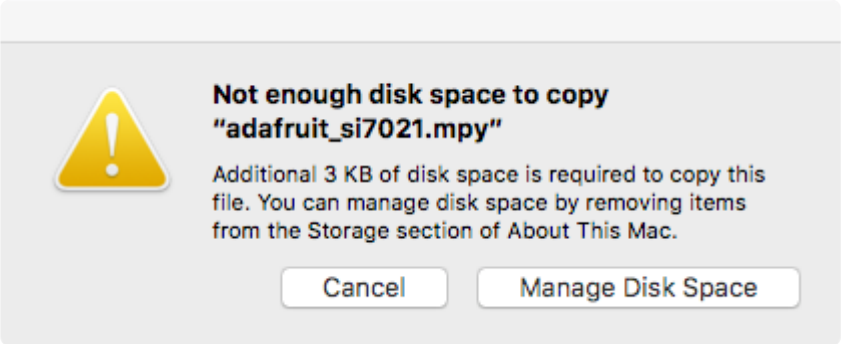
Any SAMD21-based microcontroller that does not have external flash available is considered a SAMD21 non-Express board. Non-Express boards that do not have a UF2 bootloader include the Feather M0 Basic Proto, Feather Adalogger, or the Arduino Zero.

If you are trying to erase a non-Express board that does not have a UF2 bootloader, follow these directions to reload CircuitPython using `bossac` (), which will erase and re-create CIRCUITPY.

Running Out of File Space on SAMD21 Non-Express Boards

Any SAMD21-based microcontroller that does not have external flash available is considered a SAMD21 non-Express board. This includes boards like the Trinket M0, GEMMA M0, QT Py M0, and the SAMD21-based Trinkey boards.

The file system on the board is very tiny. (Smaller than an ancient floppy disk.) So, its likely you'll run out of space but don't panic! There are a number of ways to free up space.



Delete something!

The simplest way of freeing up space is to delete files from the drive. Perhaps there are libraries in the lib folder that you aren't using anymore or test code that isn't in use. Don't delete the lib folder completely, though, just remove what you don't need.

The board ships with the Windows 7 serial driver too! Feel free to delete that if you don't need it or have already installed it. It's ~12KiB or so.

Use tabs

One unique feature of Python is that the indentation of code matters. Usually the recommendation is to indent code with four spaces for every indent. In general, that is recommended too. However, one trick to storing more human-readable code is to use a single tab character for indentation. This approach uses 1/4 of the space for indentation and can be significant when you're counting bytes.

On MacOS?

MacOS loves to generate hidden files. Luckily you can disable some of the extra hidden files that macOS adds by running a few commands to disable search indexing and create zero byte placeholders. Follow the steps below to maximize the amount of space available on macOS.

Prevent & Remove MacOS Hidden Files

First find the volume name for your board. With the board plugged in run this command in a terminal to list all the volumes:

```
ls -l /Volumes
```

Look for a volume with a name like CIRCUITPY (the default for CircuitPython). The full path to the volume is the /Volumes/CIRCUITPY path.

Now follow the steps from this question () to run these terminal commands that stop hidden files from being created on the board:

```
mdutil -i off /Volumes/CIRCUITPY
cd /Volumes/CIRCUITPY
rm -rf .{,_.}{fseventsd,Spotlight-V*,Trashes}
mkdir .fseventsd
touch .fseventsd/no_log .metadata_never_index .Trashes
cd -
```

Replace /Volumes/CIRCUITPY in the commands above with the full path to your board's volume if it's different. At this point all the hidden files should be cleared from the board and some hidden files will be prevented from being created.

Alternatively, with CircuitPython 4.x and above, the special files and folders mentioned above will be created automatically if you erase and reformat the filesystem. WARNING: Save your files first! Do this in the REPL:

```
>>> import storage
>>> storage.erase_filesystem()
```

However there are still some cases where hidden files will be created by MacOS. In particular if you copy a file that was downloaded from the internet it will have special metadata that MacOS stores as a hidden file. Luckily you can run a copy command from the terminal to copy files without this hidden metadata file. See the steps below.

Copy Files on MacOS Without Creating Hidden Files

Once you've disabled and removed hidden files with the above commands on macOS you need to be careful to copy files to the board with a special command that prevents future hidden files from being created. Unfortunately you cannot use drag and drop copy in Finder because it will still create these hidden extended attribute files in some cases (for files downloaded from the internet, like Adafruit's modules).

To copy a file or folder use the -X option for the cp command in a terminal. For example to copy a file_name.mpy file to the board use a command like:

```
cp -X file_name.mpy /Volumes/CIRCUITPY
```

(Replace file_name.mpy with the name of the file you want to copy.)

Or to copy a folder and all of the files and folders contained within, use a command like:

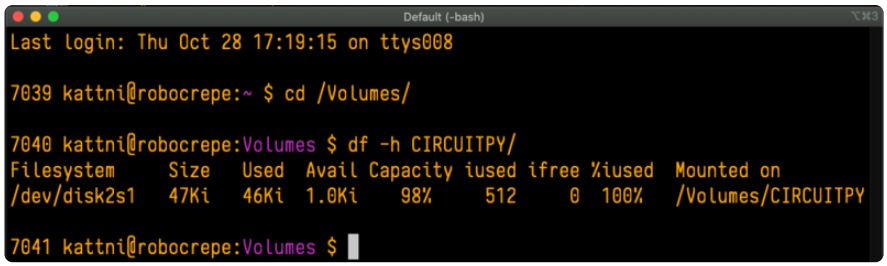
```
cp -rX folder_to_copy /Volumes/CIRCUITPY
```

If you are copying to the lib folder, or another folder, make sure it exists before copying.

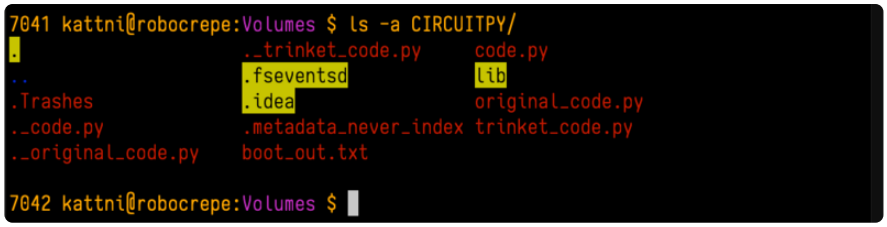
```
# if lib does not exist, you'll create a file named lib !
cp -X file_name.mpy /Volumes/CIRCUITPY/lib
# This is safer, and will complain if a lib folder does not exist.
cp -X file_name.mpy /Volumes/CIRCUITPY/lib/
```

Other MacOS Space-Saving Tips

If you'd like to see the amount of space used on the drive and manually delete hidden files here's how to do so. First, move into the Volumes/ directory with `cd /Volumes/`, and then list the amount of space used on the CIRCUITPY drive with the `df` command.

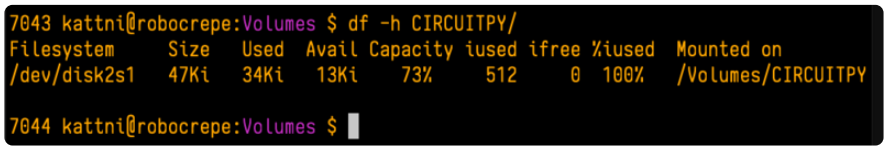


That's not very much space left! The next step is to show a list of the files currently on the CIRCUITPY drive, including the hidden files, using the `ls` command. You cannot use Finder to do this, you must do it via command line!



There are a few of the hidden files that MacOS loves to generate, all of which begin with a ._ before the file name. Remove the ._ files using the `rm` command. You can remove them all once by running `rm CIRCUITPY/._*`. The `*` acts as a wildcard to apply the command to everything that begins with ._ at the same time.

Finally, you can run `df` again to see the current space used.



Nice! You have 12Ki more than before! This space can now be used for libraries and code!

Device Locked Up or Boot Looping

In rare cases, it may happen that something in your code.py or boot.py files causes the device to get locked up, or even go into a boot loop. A boot loop occurs when the board reboots repeatedly and never fully loads. These are not caused by your everyday Python exceptions, typically it's the result of a deeper problem within CircuitPython. In this situation, it can be difficult to recover your device if CIRCUITPY is not allowing you to modify the code.py or boot.py files. Safe mode is one recovery option. When the device boots up in safe mode it will not run the code.py or boot.py

scripts, but will still connect the CIRCUITPY drive so that you can remove or modify those files as needed.

The method used to manually enter safe mode can be different for different devices. It is also very similar to the method used for getting into bootloader mode, which is a different thing. So it can take a few tries to get the timing right. If you end up in bootloader mode, no problem, you can try again without needing to do anything else.

For most devices:
Press the reset button, and then when the RGB status LED blinks yellow, press the reset button again. Since your reaction time may not be that fast, try a "slow" double click, to catch the yellow LED on the second click.

For ESP32-S2 based devices:
Press and release the reset button, then press and release the boot button about 3/4 of a second later.

Refer to the diagrams above for boot sequence details.

"Uninstalling" CircuitPython

A lot of our boards can be used with multiple programming languages. For example, the Circuit Playground Express can be used with MakeCode, Code.org CS Discoveries, CircuitPython and Arduino.

Maybe you tried CircuitPython and want to go back to MakeCode or Arduino? Not a problem. You can always remove or reinstall CircuitPython whenever you want! Heck, you can change your mind every day!

There is nothing to uninstall. CircuitPython is "just another program" that is loaded onto your board. You simply load another program (Arduino or MakeCode) and it will overwrite CircuitPython.

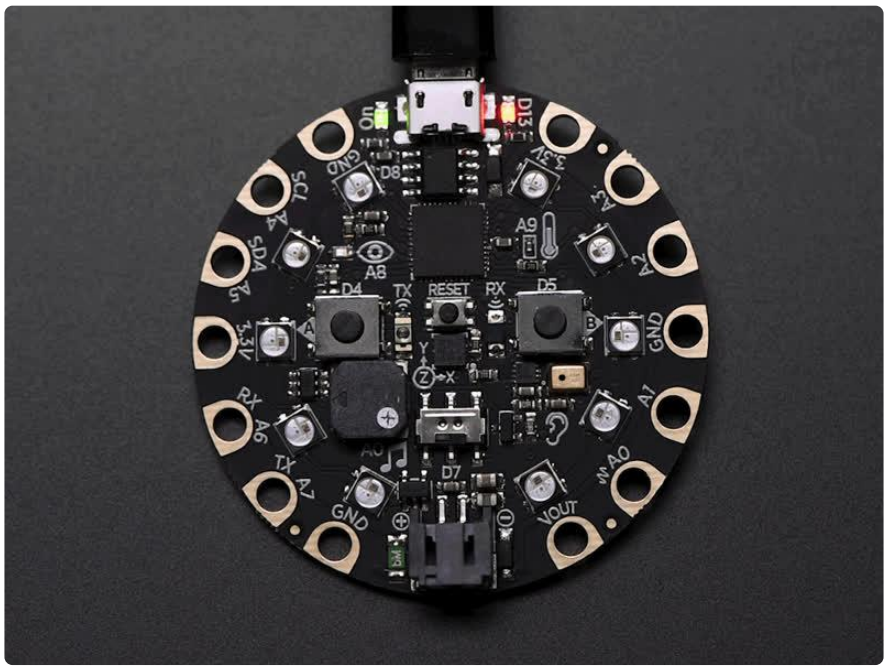
Backup Your Code

Before replacing CircuitPython, don't forget to make a backup of the code you have on the CIRCUITPY drive. That means your code.py any other files, the lib folder etc. You may lose these files when you remove CircuitPython, so backups are key! Just drag the files to a folder on your laptop or desktop computer like you would with any USB drive.

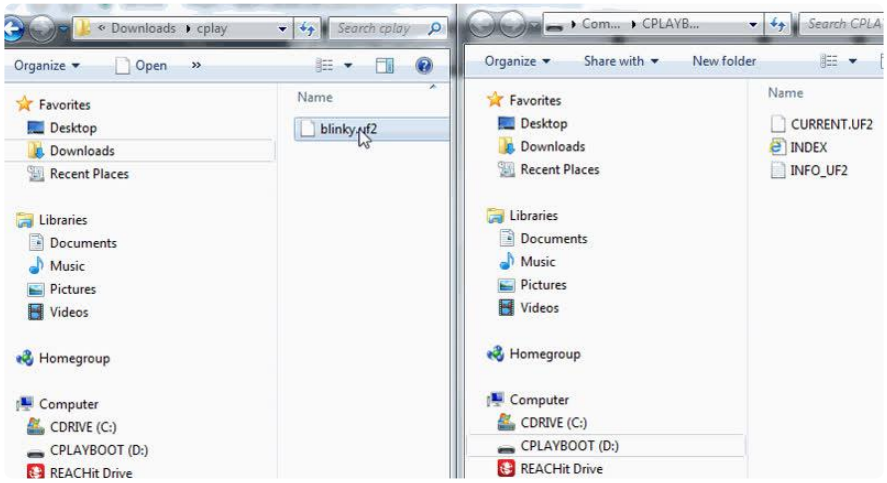
Moving Circuit Playground Express to MakeCode

On the Circuit Playground Express (this currently does NOT apply to Circuit Playground Bluefruit), if you want to go back to using MakeCode, it's really easy. Visit makecode.adafruit.com () and find the program you want to upload. Click Download to download the .uf2 file that is generated by MakeCode.

Now double-click your CircuitPython board until you see the onboard LED(s) turn green and the ...BOOT directory shows up.



Then find the downloaded MakeCode .uf2 file and drag it to the CPLAYBOOT drive.



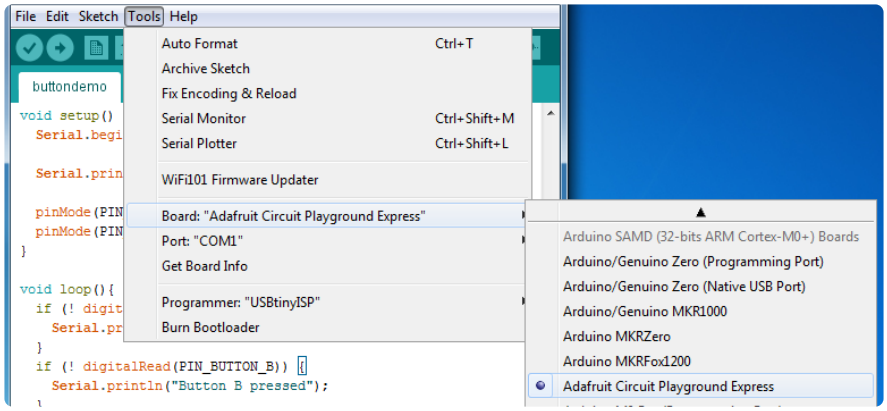
Your MakeCode is now running and CircuitPython has been removed. Going forward you only have to single click the reset button to get to CPLAYBOOT. This is an idiosyncrasy of MakeCode.

Moving to Arduino

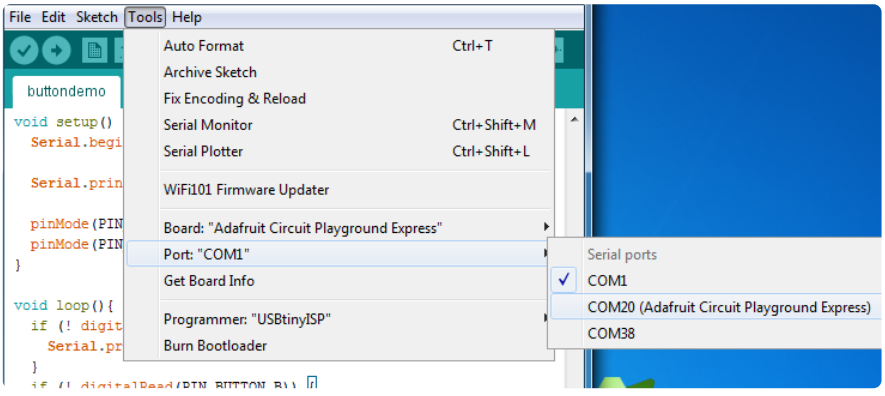
If you want to use Arduino instead, you just use the Arduino IDE to load an Arduino program. Here's an example of uploading a simple "Blink" Arduino program, but you don't have to use this particular program.

Start by plugging in your board, and double-clicking reset until you get the green onboard LED(s).

Within Arduino IDE, select the matching board, say Circuit Playground Express.



Select the correct matching Port:



Create a new simple Blink sketch example:

```
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(13, OUTPUT);
}
```

```
// the loop function runs over and over again forever
void loop() {
  digitalWrite(13, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);              // wait for a second
}
```

Make sure the LED(s) are still green, then click Upload to upload Blink. Once it has uploaded successfully, the serial Port will change so re-select the new Port!

Once Blink is uploaded you should no longer need to double-click to enter bootloader mode. Arduino will automatically reset when you upload.

CircuitPython Essentials

You've gone through the Welcome to CircuitPython guide. You've already gotten everything setup, and you've gotten CircuitPython running. Great! Now what? CircuitPython Essentials!

There are a number of core modules built into CircuitPython and commonly used libraries available. The Essentials guide will introduce you to these and show you an example of how to use each one.

It's time to get started learning the CircuitPython Essentials ()!

How Do I Learn Python?

A question that is often asked is, "How do I learn Python?" Obviously, there's no canonical answer because everyone learns in different ways. To that end, though, this page contains a list of links that can at least help you get started.

Python for Beginner Non-Programmers

Links in this section are for folks who are entirely new to programming and are interested in beginning with Python.

- Getting started with Python ()
- Python Beginner's Guide: Overview ()
- Python learning resources for non-programmers ()

Python for Programmers

Links in this section are for folks who are new to Python, but not new to programming in general.

- Python learning resources for programmers ()
- Python resources for programmers available in learner's native languages ()

Python FAQs

These are FAQs from the official Python documentation.

Basic FAQs:

- General Python FAQ ()
- Python on Windows FAQ ()

Advanced FAQs:

- Design and History FAQ ()

Python Success Stories

You may be wondering, "What can I do with Python?" The Python folks have provided an extensive list of real-life applications of Python ().

You can also refer to the Adafruit Learning System () for dozens upon dozens of CircuitPython programs used in real projects. Feel free to reuse any code you wish, it is Open Source.

Final Note

This list of resources is not exhaustive by any stretch of the imagination. It is meant to provide you with a place to start.

If you're trying to learn Python, and you're having trouble, don't be discouraged! Everyone learns in different ways and at their own pace. If you're working through something, and the way it's presented not sinking in, don't be afraid to look for a different resource. Be sure to stick with it until you find your own style!

Archive

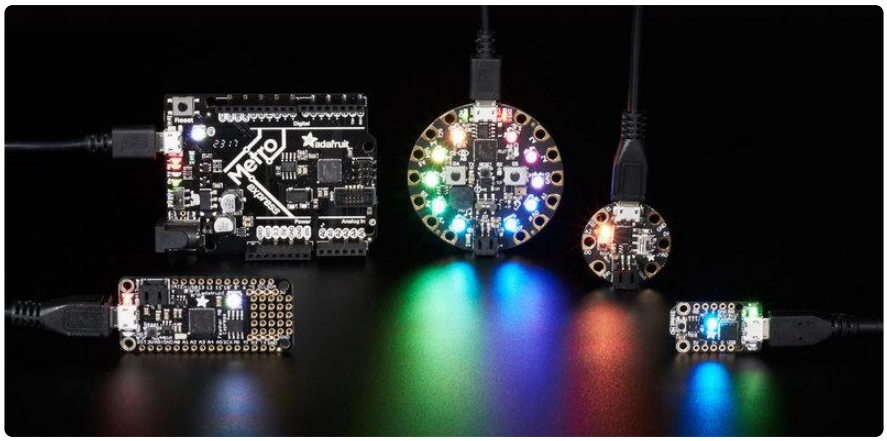
All pages in this category contain concepts that are no longer relevant or supported!

As CircuitPython development continues, the inevitable deprecation of features occurs, resulting in guide pages that have information that is no longer relevant or supported. Adafruit prefers to preserve all guide pages. Therefore pages are not deleted, but instead, are marked as deprecated.

The pages nested under this page are no longer supported. You may refer to them, but please understand that Adafruit no longer supports the concepts within them.

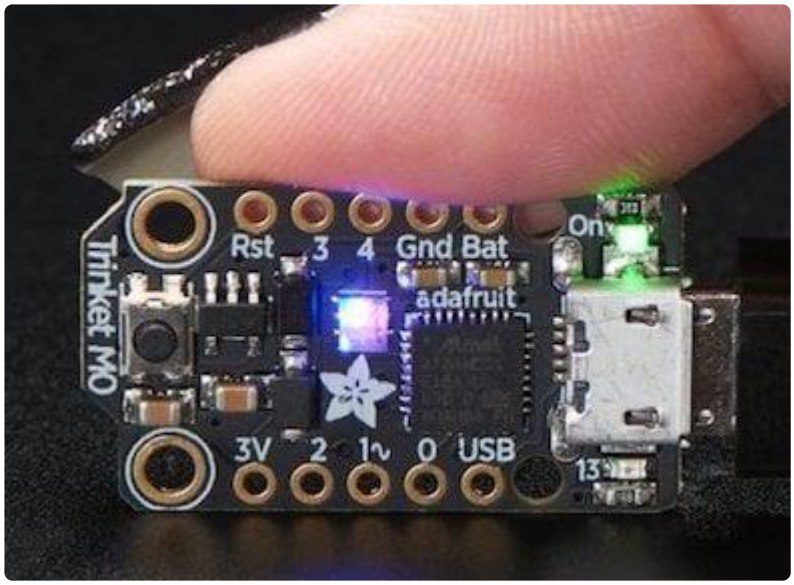
CircuitPython M0 Hardware

Other than Circuit Playground (which IS recommended!), it is no longer recommended to purchase M0 (ATSAMD21) boards for use with CircuitPython projects. There are now newer microcontrollers available at the same price-point that are faster and have more memory. The information available on this page is still valid for the boards listed.



Now it's time to do something great with what you've learned! Every CircuitPython board is perfect for projects. However, each one excels in different areas. This page will provide you with some details about each board, and highlight Learn guides where each one is used. You can try these out or get ideas for your own project!

Trinket M0



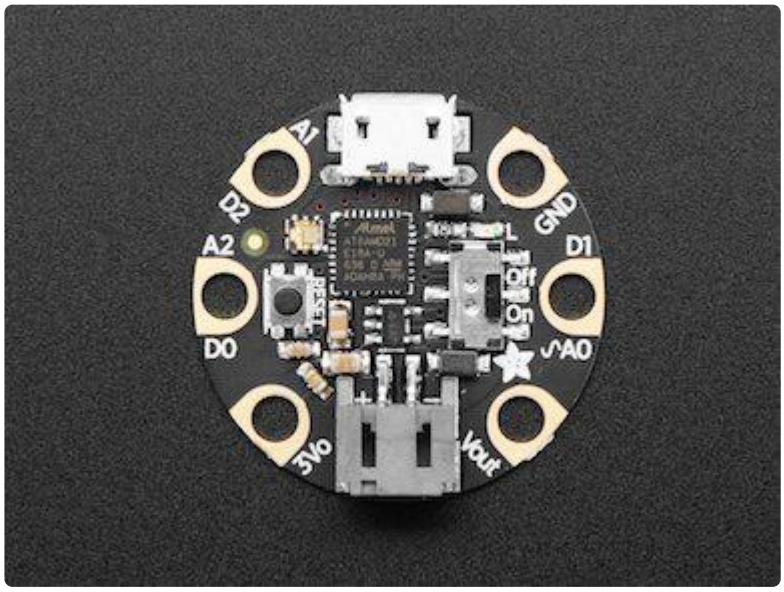
The Adafruit Trinket M0 () is the smallest CircuitPython board Adafruit carries. But don't let that fool you! It's a tiny board with a lot of power. Adafruit wanted to design a microcontroller board that was small enough to fit into any project, and low cost enough to use without hesitation. Planning to test a proof of concept and need a CircuitPython board to throw in? Not ready to disassemble the project you worked so hard to design to extract the board you used last time? Trinket M0 has you covered. It's the lowest cost CircuitPython board available but it easily holds its own with the bigger boards!

Trinket M0 ships with CircuitPython and comes with demo code already on the board. You can open and edit the main.py file found on the CIRCUITPY drive to get started,

or create your own! The Trinket M0 guide () gives you everything you need to know about your board. Check out the CircuitPython () section to find a huge list of examples to try.

You can use Trinket M0 to make the Chilled Drinkibot () which uses the Trinket to control thermoelectric cooling of a beverage. Or build a spooky Halloween project that turns your candy bucket into a Screaming Cauldron ()!

Gemma M0

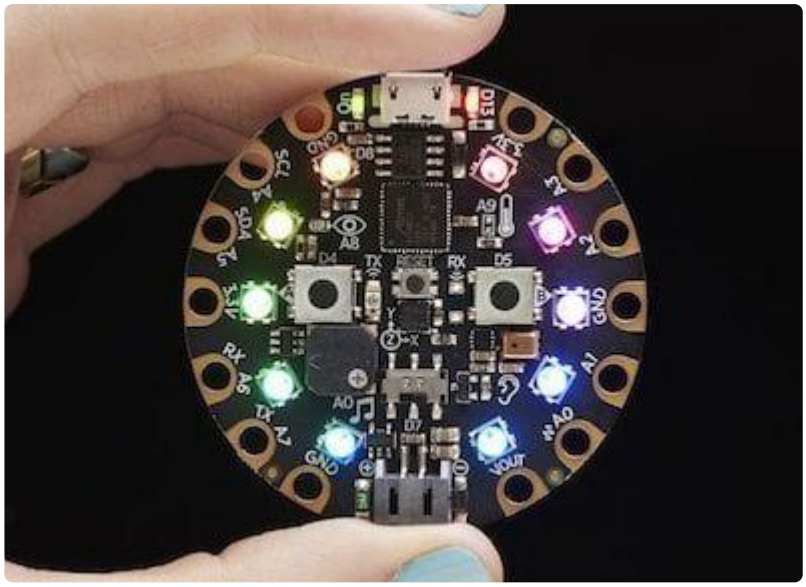


The Adafruit Gemma M0 () is a tiny CircuitPython board with just enough built-in to build many simple projects. It's designed to be worked into your wearable projects, with big holes around the outside for sewing (and they're alligator clip friendly too!). Gemma M0 will super-charge your wearables and is easier to use than ever. There are capacitive touch pads, an on-off switch and an RGB DotStar LED built right into the board so there's plenty you can do without adding a thing. Add conductive thread and LEDs and you'll have a blinky wearable in no time!

Like Trinket, Gemma M0 ships with CircuitPython and has demo code already on the board. You can open and edit the main.py file on the CIRCUITPY drive, or create your own! The Gemma M0 guide () shows you all the info about your board, and has a great list of CircuitPython examples () to try out.

Use Gemma M0 to create a pair of Clockwork Goggles () with fun light patterns on NeoPixel rings. Or accessorise with this 3D printed Sheikah Pendant () to add a bit of light to your next costume!

Circuit Playground Express

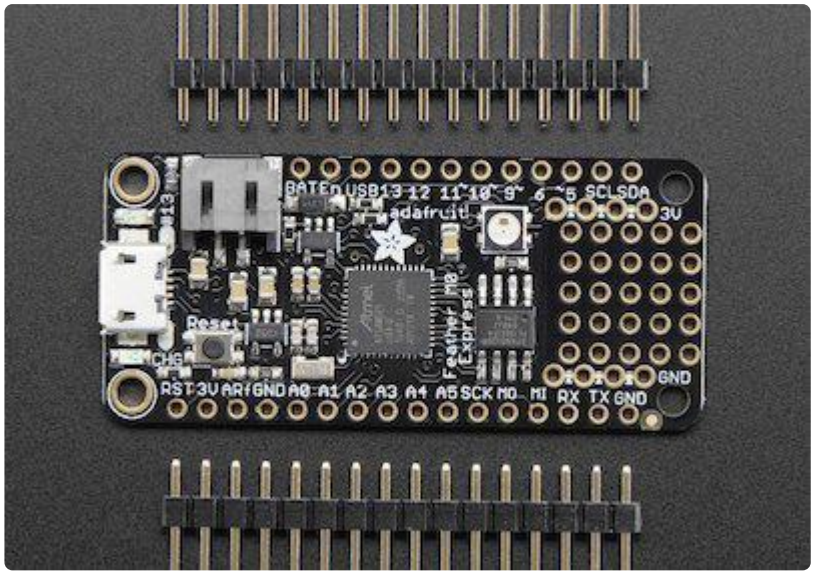


The Adafruit Circuit Playground Express () is the next step towards a perfect introduction to electronics and programming. It's packed full of sensors, LEDs, buttons and switches, and it's super easy to get started with! This board is super versatile. Whether you're new to electronics and programming, or a seasoned veteran, Circuit Playground Express is an amazing board to work with. With so much built into the board, you can learn how different types of electronics work and learn to program them all without purchasing any other parts. All you need is a USB cable and the board! But, that's just the beginning. Many of the pads around the outside of the board function in multiple ways allowing you to wire other things to the board. For example, you could wire up a servo or a potentiometer. The possibilities are endless!

The Circuit Playground Express guide () has tons of information on all the fantastic features of the board. The CircuitPython section () of the guide has an extensive list of examples using the built-in features of the CircuitPython and the board. There is also a section called Python Playground () with more demos and a Drum Machine project () to try out.

You can turn your Circuit Playground Express into a capacitive touch tone Piano in the Key of Lime () using the touch pads on the board. Use the built in accelerometer to make a UFO Flying Saucer () complete with lights and alien sounds using your board, and some extra supplies from around the house or a 3D printed saucer!

Feather M0 Express

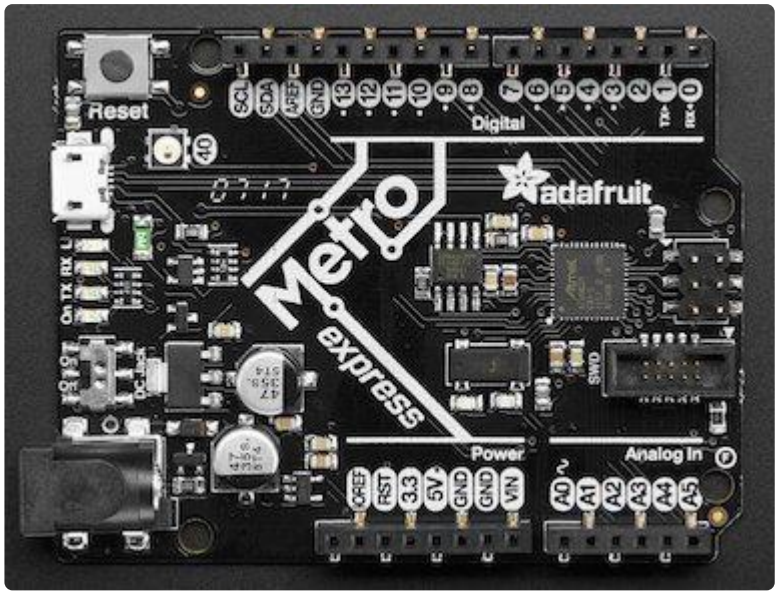


The Adafruit Feather M0 Express () is the first Feather designed specifically for CircuitPython. It's part of a line of Adafruit Feather development boards () designed to work standalone or stacked, and is powered by USB or lithium ion battery so it works for both stationary and on-the-go projects. Feather M0 Express comes with two headers for use with a solderless breadboard, or you can solder wires directly to the pins on the board. This allows for prototyping while you're working on your project and permanent installation when you're ready. One of the things that makes the Feather M0 Express amazing is the the huge array of boards called Featherwings () which are designed to fit right on the Feather. There are CircuitPython libraries for many of these boards and more are being written all the time.

Feather M0 Express ships CircuitPython-ready with the UF2 bootloader installed and ready for you to install CircuitPython when you receive your board. Create your first program, save it to the board, and off you go! The Feather M0 Express guide () has all the details about your board, and a CircuitPython section () to get you started.

Feather M0 Express can be used to power all kinds of projects. Build a CircuitPython Painter () POV LED wand using 3D printed parts and DotStar strips. Create an engraved edge-lit LED Acrylic Sign () with NeoPixels. There are guides to go with the Featherwings that explain how to use them with CircuitPython, like the OLED Display ( ) and Adalogger Featherwing ().

Metro M0 Express



Metro M0 Express () is the first Metro board designed to work with CircuitPython. This is not a beginner board. If you're just getting started, Adafruit recommends one of the previous boards. It has a lot of the same features as Feather M0 Express, as well as some development specific features (like the SWD port built in!). The Metro M0 Express is designed to work with the Arduino form-factor, so if you've already got Arduino shields, this board would be great for you. There are CircuitPython libraries for some shields already. It has 25 GPIO pins (the most of any of these boards!) so it's great if you're looking for a lot of options.

Metro M0 Express ships CircuitPython-ready with the UF2 bootloader installed and is ready for you to install CircuitPython when you receive your board. Create your first program, save it to the board, and you're good to go! The Metro M0 Express guide () gives you all the details about your board, and the CircuitPython section () is available to get you started.

All sensors and breakout boards with CircuitPython libraries will work with the Metro M0 Express running CircuitPython. Find the guide for your sensor and follow the guide to find out how to wire it up. There are a ton of options available.

What's Next?

Now you're ready to jump into some more Learn Guides or simply get started with a brand new project. Great job, and good luck!

CircuitPython for ESP8266

We are no longer supporting CircuitPython on ESP8266. This page is for historical purposes only. There is no guarantee that the instructions will continue to work.

Why are we dropping support for ESP8266?

CircuitPython on ESP8266 has not been a great experience for users. Its hard to send files to the device because it has no native USB, and you can quickly run out of RAM (there's less than you think once the networking stack is added). We've decided to only use ESP as a co-processor. Specifically, the ESP32 because it has good TLS/SSL support which is now essential for even the most basic interactions.

If you'd like to use ESP8266, please keep to version 3.x with the knowledge that it isn't supported. You can also check out MicroPython for ESP8266 which is still supported!

If you'd like to add WiFi support, check out our guide on ESP32/ESP8266 as a co-processor. ()

About ESP8266 for CircuitPython (3.x)

We have two 'strains' of CircuitPython, the primary one is the ATSAMD21/51-based boards that have native USB connectivity. Native USB means that the board can show up as a disk drive called `CIRCUITPY` and hold all your files on it.

There's also CircuitPython for boards like the ESP8266 and nRF52832, these are really nice chips with WiFi and Bluetooth, respectively, built in. But they do not have native USB! That means there is no way for the chip to appear as a disk drive. You can still use them with CircuitPython but its a lot tougher, so we don't recommend them for beginners. Here's what you have to know about using non-native chips for CircuitPython:

- You only get a REPL connection! No HID keyboard/mouse or other USB interface
- No disk drive for drag-n-drop file moving, files must be moved via a special tool such as ampy that 'types' the file in for you via the REPL

- Loading CircuitPython requires command line tools

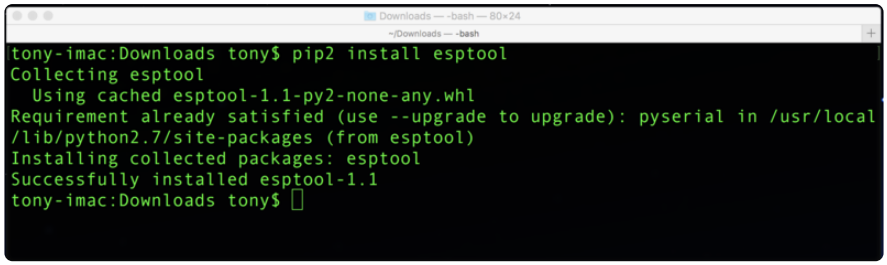
Installing CircuitPython on the ESP8266

To use CircuitPython with the ESP8266 you'll need to first flash it with the latest firmware.

Download esptool

First install the `esptool.py` software which enables firmware flashing on the ESP8266. The easiest way to install this tool is from Python's pip package manager. If you don't have it already you'll need to install Python 2.7 () (make sure you check the box to put Python in your system path when installing on Windows) and then run the following command in a terminal: `pip install esptool`

Note on Mac OSX and Linux you might need to run the command as root with sudo, like: `sudo pip install esptool`



If you receive an error that `esptool.py` only supports Python 2.x try running again with the pip2 command instead of pip (likely your system is using Python 3 and the pip command is getting confused which version to use).

Download Latest CircuitPython Firmware

Next download the latest CircuitPython ESP8266 firmware file:

Download Latest ESP8266 Huzzah Firmware (v3.1.2)

Get ESP8266 Ready For Bootloading

Now you'll need to put the ESP8266 into its firmware flashing mode. Each ESP8266 board is slightly different:

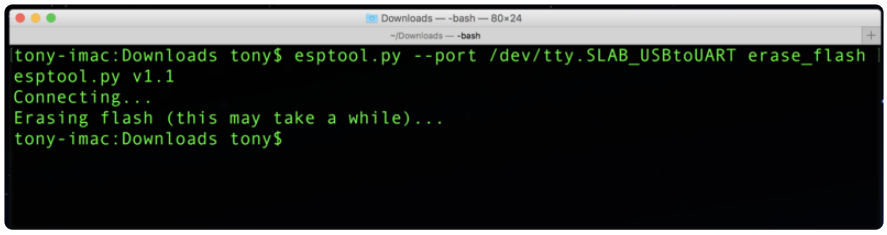
- For a raw ESP8266 module you'll need to wire up buttons to ground for the GPIO0 and RESET pins. Hold the GPIO0 button down (or connect the line to ground) and while still holding GPIO0 to ground press and release the RESET button (or connect and release the line from ground), then release GPIO0.
- For the HUZZAH ESP8266 breakout () buttons for GPIO0 and RESET are built in to the board. Hold GPIO0 down, then press and release RESET (while still holding GPIO0), and finally release GPIO0.
- For the Feather HUZZAH ESP8266 () you don't need to do anything special to go into firmware flashing mode. This board is built to detect when the serial port is opened for flashing and automatically configure the ESP8266 module to receive firmware. Be sure to first install the SiLabs CP210x driver () on Windows and Mac OSX to make the board's serial port visible! On Windows you want the normal VCP driver, not the 'with Serial Enumeration' driver.

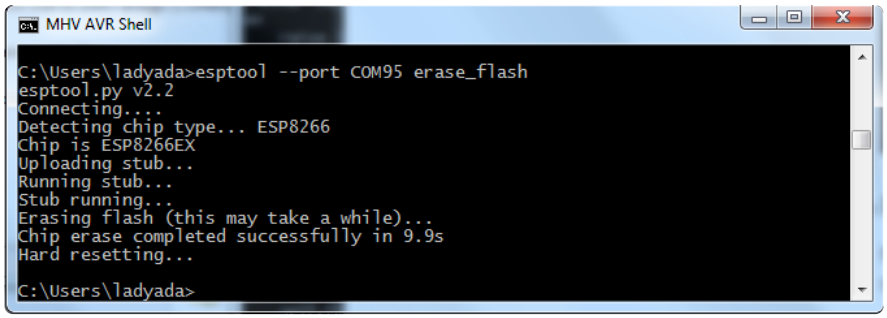
Erase ESP8266

It's recommended to erase the entire flash memory of the ESP8266 board before uploading firmware. Run the following command in a terminal to perform this erase:

```
esptool.py --port ESP8266_PORTNAME erase_flash
```

Where `ESP8266_PORTNAME` is the path or name of the serial port that is connected to the ESP8266. The exact name of the device varies depending on the type of serial to USB converter chip so you might need to look at the serial ports with and without the device connected to find its name.



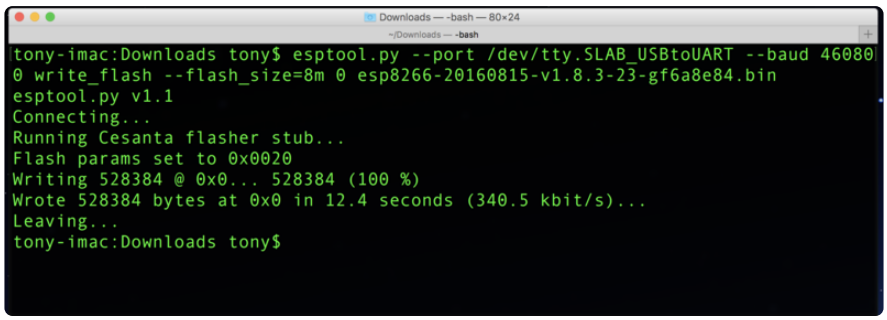


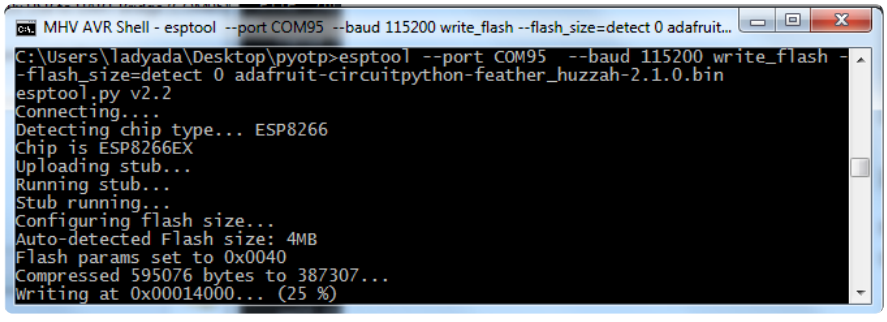
Program ESP8266

Now put the ESP8266 back into firmware flashing mode and run the following command to load the downloaded firmware file:

```
esptool.py --port ESP8266_PORTNAME --baud 115200 write_flash --flash_size=detect 0
firmware.bin
```

Again set `ESP8266_PORTNAME` to the path or name of the serial port that is connected to the ESP8266. In addition set firmware.bin to the name or path to the firmware file you would like to load.





Once the tool finishes flashing the firmware (you'll usually see a blue light on the ESP8266 module flashing during this process) press the RESET button on the ESP8266 board or disconnect and reconnect it to your computer. You should be all set to start using the latest CircuitPython firmware on the board!

Note that if you see an error that "detect is not a valid flash_size parameter" you might be using an older version of esptool.py. To upgrade to the latest version run the following command `pip install --upgrade esptool`

Upload Libraries & Files Using Ampy!

The biggest difference you'll find with ESP8266 is that you need to use a special tool to move files around. Check out `ampy` by reading this guide. It's about MicroPython but CircuitPython is nearly identical so the the overall installation and usage is identical!

Learn how to use ampy to move files on your ESP8266

Other Stuff To Know!

- The REPL works as you'd expect, so check out that introductory page.
- File storage is in the same chip as CircuitPython so if you update, you may lose your files! Keep backups.
- Libraries and API are also the same as for other CircuitPython boards.
- Note that the ESP8266 does not have a ton of pins available, and only one analog input with 0-1.0V range. There is no UART port available (it's the one used for the REPL!)
- There are no analog outputs.
- For SPI and I2C, you can use them! But you will need to use bitbangio () to create the bus objects